Chapter 3
Data and Query Model

In this chapter, we review some of the key concepts in databases to help readers better understand the later parts of this book. We begin the chapter with the notion of **data model**, which can be seen as a syntax for data, similar to a syntax defined using a grammar for natural and programming languages. It provides an abstraction for describing the constraints on data, querying data and data manipulation. We introduce the notion of **conceptual model**, which describes the design structure of a database. We then describe two common types of data models: (1) **relational model**, which organizes data into collections of two-dimensional tables called "relations"; and (2) **graph model**, which organizes data as a graph. We introduce the query languages for both relational and graph models using examples.

We finally take a brief look at the journey of a query within DBMS and some of the components such as **storage structures**, **query evaluation and optimization**, which are responsible for evaluating the query over a database instance and retrieving the results. Understanding these components may provide some insight on the cost of queries and the evaluation of NLIDBs (Chapter 5).

3.1 Conceptual Models

Even though data is ultimately recorded as a sequence of bytes in the secondary storage, working with data at such a low physical level is tedious. On the other hand, structured data often conforms to a schema that describes both the **type** and the **structure** of data. Assigning a type, for example, to each column of a table ensures that the values that are being stored are syntactically correct. The type information also helps in parsing or interpreting data by interfaces that are built on top. The structure, on the other hand, describes how to organize different but interrelated pieces of data (e.g., *Bill Gates*, *Harvard University*, *1955* and *Microsoft Corporation*) and expression their relationships.

At a high level, the design structure of a database may be described using a conceptual modeling scheme such as **entity-relationship** (**ER**) diagram or **universal modeling language** (**UML**). These are not considered data models but design methodologies to more formally describe the structure of and the relationships between different data components.

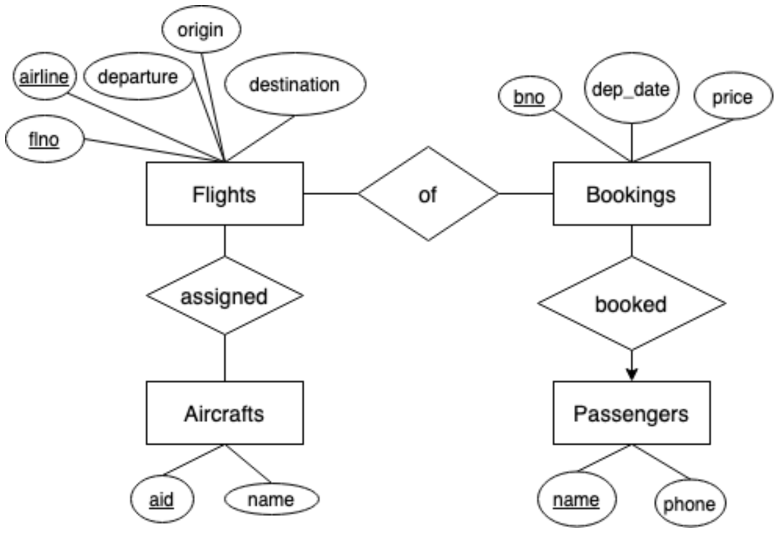


Fig. 3.1 An example ER diagram

3.1.1 Entity relationship model

The two basic components of an ER model [52] are **entities** and **relationships**. Entities describe objects or concepts that can be distinguished from other objects using a set of attributes. Examples are `flights`, `passengers`, `student`, `course`, `teacher`, etc. Relationships describe the fact that one entity is in relationship with another entity. Some examples are *Flight UA541* is using *Boeing 737 aircraft*, *Davood* is teaching *CMPUT 291*, *Mary* is booked on *Flight UA541* departing on *December 20, 2021*.

Figure 3.1 depicts the ER diagram for a flights database. As can be seen, the database consists of four entities: `Flights`, `Bookings`, `Aircrafts` and `Passengers`, each shown as a rectangle in the diagram. It also contains the relationships linking those entities: `assigned`, `of`, and `booked`, each shown as a diamond in the diagram. Entities and relationships can also have **attributes**. As shown in Figure 3.1, the `Flights` entity has attributes `flno`, `airline`, `departure`, `origin` and `destination`, shown as ovals in the diagram. The underlined attributes, referred to as **keys**, are unique for each entity (e.g., `airline` and `flno` together is unique for each flight and `bno` is unique for each booking). The arrow from Bookings to Passengers indicates that each booking is made by one passenger, whereas no such constraint is defined over other relationships. The basic ER model may be extended with other components to describe more complex relationships (e.g., ternary and isa hierarchy) and constraints (e.g., key and participation constraints). A detailed textbook coverage of the ER model and the relevant research work is provided by Thaleheim [309].

3.1.2 UML

The initial design goal of UML [271] was to model software. It is an accumulation of multiple software engineering methodologies that have been in use before their formation as a unified language. The language offers different modeling diagrams, each for a different aspect of software development, such as state diagram, activity diagram, use case diagram, deployment diagram, etc. For representing entities, UML offers the **class diagram**, which describes objects and their relationships similar to ER. Unlike ER, an object

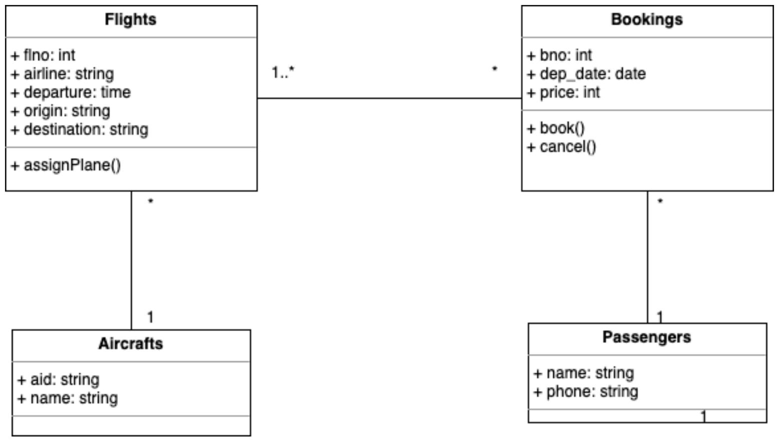


Fig. 3.2 A UML class diagram

description may include not only the attributes but also the methods that operate on them. For example, a class diagram for our flight database depicted in Figure 3.1 may include four classes `flights`, `bookings`, `aircrafts` and `passenger`. The class diagram may show the attributes and methods of each class as well as the relationships between classes as shown in Figure 3.2. More information about database modeling using UML can be found elsewhere (e.g., [308]).

3.1.3 Ontology

Ontologies may add a semantic layer to a conceptual model by defining the meaning of terms and concepts and formalizing their relations. Generally each item in an ontology is either a class or an instance of a class. For example in our flight database, passengers and aircrafts define some classes and John and Boeing 757 are instances of these classes respectively. A common relationship type is subclass-of. For example, we can say that `passenger` is a person and `aircraft` is a thing. Two other relationship types are equivalent and disjoint. For example, in airline reservation, one may say that `passenger` and `customer` are equivalent, meaning that they contain the same set of instances. The major strength of ontologies is in describing the relationships between different classes and allowing us to make inference. For example, knowing that person and thing are disjoint, we can say that `passenger` and `aircraft` are also disjoint; or given that each customer has a home town, each passenger must have a home town. A tutorial style introduction to ontologies is given by Jepsen [157] and the role of ontologies in database design is studied by a few others (e.g., [300]).

Ontolgies can play an important role in interpreting natural language questions and mapping them to formal queries over a database. For example, consider a natural language question that references customers (and not passengers) and a database that has a table called Passengers. With an ontology that states passenger is equivalent to customer, one can link the question to the passengers table; this may not be straightforward without an ontology. Similarly, an ontology that lists some instances of classes (e.g., John is a passenger and Boeing 757 is an aircraft) can help in mapping question words to database elements [272, 186].

3.2 Relational Model

The **relational model**, as introduced by Ted Codd [69], is built around the simple but mathematically sound concept of **relations** (or **tables**) for describing the structure of data and an algebra and a calculus for expressing queries. Because of its simplicity, the model is easily accessible to a broad range of end users; due to its foundation on the set theory, relations can be queried using powerful declarative query languages. The model provides means for describing integrity constraints in the form of **domain constraints**, **primary key** and **foreign key** constraints, and **functional dependencies**, to be described shortly.

airline	flno	origin	destination	dep	dist (km)	dur (min)	aid
AC	162	Edmonton	Toronto	7:30	2690	218	a1
AC	163	Toronto	Edmonton	8:30	2690	247	a2
UA	1274	New York	Seattle	6:45	3880	363	a3
UA	541	Seattle	Denver	9:03	1670	163	a3
UA	208	Denver	New York	13:30	2580	218	a4
UA	714	San Jose	Seattle	10:40	1129	140	a5
UA	1526	Seattle	San Jose	15:10	1129	125	a3

(a) Flights

passenger	airline	flno	dep_date	price
Davood	AC	162	2021-08-14	200
Davood	AC	163	2021-08-17	200
Drago	UA	1274	2021-11-14	150
Drago	UA	541	2021-11-18	150
Drago	UA	208	2021-11-18	150
Yunyao	UA	714	2021-09-02	160
Yunyao	UA	1526	2021-09-04	160

(b) Bookings

aid	name
a1	Airbus A220-300
a2	Boeing 737 MAX 8
a3	Boeing 737
a4	Boeing 757
a5	Airbus A319

(c) Aircrafts

name	phone
John	416-111-1111
Mary	650-222-2222
Yunyao	408-333-3333
Drago	203-444-4444
Davood	780-555-5555

(d) Passengers

Fig. 3.3: Relational database example

Figure 3.3 shows an example relational database. As can be seen, generally each row describes an entity (e.g., *Flight AC162*) or a relationship (e.g., *Davood* is booked on *Flight AC162* departing on *August 14th, 2021*). Rows that describe the same class of entities or relationships usually have the same structure and are also known as **tuples**. Tuples are grouped into tables, also known as **relations** (e.g., `Flights`, `Bookings`). We distinguish between table structure, referred to as **schema**, and table content, referred to as **instance**.

A relation schema may consist of a relation name, the names and types of the columns, and sometimes **integrity constraints**. Two commonly used constraints are **primary key** and **foreign key** constraints. The **primary key** of a table refers to a set of table attributes that is unique and is the primary means of identifying tuples in the relation (e.g., flight number `flno` in the `Flights` relation). A table can have multiple sets of unique attributes, referred to as **keys**, but each table can have only one primary key. A **foreign key** of a table is a set of attributes that refers to a unique row in another table (e.g., flight number `flno` in `Bookings` refers to a valid flight number `flno` in `Flights`). More general constraints may also be specified in creating tables. Examples of such constraints are "`the number of passengers booked in a flight cannot exceed the aircraft capacity`" and "`two airlines cannot have the same flight number`."

3.2.1 Relational Query Languages

A **database query** is a way of retrieving or manipulating information from a database. For data stored in a relational database, queries can be posed using a large selection of powerful **structured query languages**. Unlike conventional programming languages, query languages are often **declarative** where one specifies the properties that must hold or the conditions that must be met but not the detailed steps for retrieving or manipulating the information. These languages often have a limited expressive power (not necessarily Turing complete) and queries expressed in such languages are more efficient to evaluate. The wide adoption of these query languages has been a major force behind storing data in a relational database.

3.2.2 First-Order Logic

First-Order Logic(**FOL**) is the mathematical foundation beneath most relational query languages. Consider the example database in Figure 3.3, and suppose we want to find all passengers who are scheduled to fly to Seattle. This query may be expressed in first-order logic as a conjunction of two predicates: flights(x_{al}, x_{fn}, '*Seattle*', y_1, y_2, y_3, y_4) and bookings(x_p, x_{al}, x_{fl}, y_5, y_6), written as

$$\texttt{ans}(x_p) \leftarrow \texttt{flights}(x_{al}, x_{fn}, \text{`}Seattle\text{'}, y_1, y_2, y_3, y_4), \texttt{bookings}(x_p, x_{al}, x_{fl}, y_5, y_6)$$

in a rule language with **bound variables** ranging over the values of each column and the same variable names (e.g., x_{al} and x_{fn}) indicating a match in the corresponding columns. Variable x_p is a **free variable** that iterates over the result set.

Generally each query in first-order logic is described as one or more rules of the form:

$$\texttt{ans}(t) \leftarrow R_1(t_1), \ldots, R_n(t_n)$$

where $R_1, \ldots, R_n$ in the rule body are the names of relations in the database, `ans` in the rule head is the relation name the query returns and $t, t_1, \ldots, t_n$ are tuples with arities that match those of the relations `ans`, $R_1, \ldots, R_n$ respectively. Each tuple t may be written as $< x_1, \ldots, x_m >$ where each x_i can be either a free variable ranging over the values of the respective domain or a constant. In most relational query languages, the variables in the rule head t are limited to those that appear in the rule body $t_1, \ldots, t_n$. For any assignment of the free variables where the rule body evaluates to truth, the head holds and it adds a tuple to ans.

3.2.3 Relational Algebra

An algebraic perspective for writing relational queries is based on a set of unary and binary operators, each taking one or two relations as input and producing a relation as the result. Three primitive operators include **selection**, **projection** and **Cartesian product**. A selection over a relation instance r, written as $\sigma_c(r)$, selects all rows in the relation that satisfy the condition c. A projection, written as $\pi_{a_i,\ldots,a_j}(r)$, returns its argument relation with attributes that are not listed left out. Removing some columns can produce duplicates, and those duplicates are removed, making the result a relation. The Cartesian product, defined as the Cartesian product of two sets and written as $r_1 \times r_2$, combines information from two relation instances. For example, to find all airlines that fly to Edmonton, we can write:

$$\pi_{airline}\sigma_{destination='Edmonton'}Flights$$

Information from multiple tables can be joined using Cartesian product and selection, but it is often easier to use a join operator. In particular, a **natural join** of tables r_1 and r_2, written as $r_1 \bowtie r_2$, joins the two tables based on their common columns, where two tuples are joined if they have the same values in the common columns. For example, the following query finds all passengers booked on a flight from Denver to New York.

$$\pi_{passenger}\sigma_{origin='Denver'\ and\ destination='NewYork'}(Flights \bowtie Bookings)$$

With each table treated as a set of rows, tables can be combined in queries using **set operations**. For example, the following query will give flights with no bookings.

$$\pi_{airline,flno}Flights - \pi_{airline,flno}Bookings$$

Hence set operations including **set union**, **set intersection** and **set difference** are included in relational algebra. Other relational algebra operators include renaming and division. More information about these operators and composing them to formulate a query can be found in any database systems (e.g., textbook [292]).

In terms of the **complexity of queries**, for tables with N rows, it is easy to see that relational algebra operators can be all computed in $O(N^2)$ time with join being the most expensive operation. On the same basis, any relational algebra query with a fixed number of operations is polynomial on the input size. On the other hand, when considering the complexity of queries, we often make distinctions between **data complexity** and **combined complexity**. In the former, we assume the query is fixed and the complexity is expressed in terms of database size, whereas in the latter, both database size and the query vary. Given a database instance I and a query Q, the cost of evaluating the query is polynomial in $|I|$ and exponential in $|Q|$ (see [242]).

3.2.4 SQL

SQL is a structured query language and an ANSI standard for querying relational databases. The language can be seen as a syntactic sugar on top of the first-order logic. Hence it is a declarative language. However, SQL supports functions to work with different data types as well as aggregation, grouping, and result ordering; these operations are not available in first-order logic.

A basic SQL query consists of *select*, *from*, and *where* clauses with the relations to be queried listed in the from clause and the query conditions to be satisfied listed in the where clause. The select clause describes the columns to be returned. For example, the query “find all flights to Seattle” can be written in SQL as

```
Q1.
  select *
  from Flights
  where destination=‘Seattle’;
```

where * in the select clause indicates that all columns of the table Flights should be returned.

SQL also has a close relationship with relational algebra, which is considered a procedural language for querying relational databases. Firstly, SQL expressions may be mapped to relational algebra expressions inside a SQL engine, and this provides opportunities for both query optimization and efficient query evaluation. For example, a basic select-from-where query is mapped to a select-project-join (SPJ) query in relational algebra. Secondly, set operations in SQL are borrowed from relational algebra, and this can make some queries with a procedural structure easier to write.

3.2.4.1 SQL query dimensions

Since its initial version back in 1974, SQL has been evolving and has gone through a few revisions, with SQL:1999 being the fourth revision of the language. With the widespread use of the language and through those revisions, SQL has become a complex language. The language complexity may be broken down along a few dimensions, in terms of the number of joined relations, the use of aggregation and grouping, query nesting, result ordering, negation and recursion, etc. Next we discuss SQL along these dimensions using examples.

3.2.4.2 Queries on a single relation

Query Q1, as given above, is an example of simple query with one relation in the from clause. The query returns all flights to Seattle. To return only the airline and flight number, one can list those column names (as listed in Figure 3.3) in the select clause.

```
  select airline, flno
  from Flights
  where destination=‘Seattle’;
```

Multiple conditions can be expressed in the where clause using logical connectors *and*, *or* and *not*.

3.2.4.3 Queries on multiple relations

Often the data that is relevant to a query formulation resides in multiple tables, and those tables need to be joined. In many cases, data is spread over multiple tables through normalization [1] or for performance reasons, and those tables may need to be joined to formulate a query. For example, one will need to join the two tables Flights and Bookings to find the passengers who are scheduled to fly to Seattle in September 2021, given as

```
Q2.
  select passenger
  from Flights f, Bookings b
  where f.airline=b.airline
    and f.flno=b.flno
    and destination=‘Seattle’
    and strftime(‘%Y’,dep_date)=‘2021’
    and strftime(‘%m’,dep_date)=‘09’;
```

where *strftime* is a Sqlite function for extracting year and month fields. Commercial systems may differ on how such functions over data types are supported. For example, Oracle and DB2 instead provide an *extract* function for extracting year, month and day fields from a date type.

3.2.4.4 Set operations

Set operations $\cup$, $\cap$ and $-$ are supported in SQL using the keywords *union*, *intersect* and *except* respectively. These operations, which correspond to their relational algebra counterparts, make it easier to write some queries with a procedural structure. For example, the next query finds airlines that are not operating any Boeing 737 Max aircraft.

```
Q3.
  select airline
  from Flights
  except
     select airline
     from Flights f, Aircrafts a
     where f.aid=a.aid and lower(name) like ‘boeing 737 max%’;
```

[1] Database normalization is a common approach for decomposing tables to reduce or eliminate the redundancy in data.

3.2.4.5 Grouping and aggregation

Data elements sometimes need to be grouped and/or aggregated to formulate a query. For example, to find all bookings of each airline, rows in Bookings must be grouped based on airline, and those groups may further be refined based on some aggregation function or other constraints added to the *having clause* in SQL. An example is retrieving those flights that have more than 20 bookings. Grouping and aggregation adds another level of complexity to the language.

```
Q4.
  select airline
  from Flights f, Bookings b
  where f.airline=b.airline and f.flno=b.flno
  group by f.airline, f.flno
  having count(*) > 20;
```

3.2.4.6 Nested queries

In the spirit of function calls in programming languages, SQL allows one query to be nested inside another query. A query nested inside an outer query, referred to as *a subquery*, is a select-from-where expression that may return a scalar or a set. Some common use cases for subqueries include membership testing, comparing a scalar to a set, and testing the set cardinality.

The outer query may test the subquery result for membership using the *in* connective. For example, the subquery may find flights that are overbooked and the outer query may check if a passenger is in one of those overbooked flights. The outer query may compare a scalar to set using *op all* or *op some* where *op* can be $>$, $\geq$, $<$, $\leq$, $=$ and $! =$. As an example, the subquery may compute the number of flights to each destination and the outer query may find destinations with the largest number of flights (see Q5). As another use case for subqueries, the outer query may want to check if the result of a subquery is non-empty or empty, which is done using the connectors *exists* and *not exists*. For example, the subquery may find all flights of a passenger on Air Canada and the outer query may check if this set is empty (see Q6).

```
Q5.
select destination
from Flights
group by destination
having count(*) >= all (select count(*)
                        from Flights
                        group by destination);
Q6.
select passenger
from Bookings p
where not exists (select *
                  from Bookings b
                  where p.passenger=b.passenger and airline='AC');
```

3.2.4.7 Ordering and top-k

The results may need to be ordered and a selection may be done on the ordered results. For example, we may want to order the results based on the number of bookings or find top five passengers with the largest number of bookings. Query *Q7* finds top three airlines with the largest number of bookings, and *Q8* sorts the aircrafts based on the average distance travelled.

```
Q7.
  select airline
  from Bookings
  group by airline
  order by count(*) desc
  limit 3;
Q8.
  select aid, name
  from Aircrafts a, Flights f
  where a.aid=f.aid
  group by aid, name
  order by avg(dist);
```

3.2.4.8 Recursion, negation and null values

Suppose we want to find all pairs of cities S and T such that a passenger can get from S to T using any number of flights. Q9 shows a typical example of recursion where the query refers to its own result to construct the full result. Recursion in SQL is defined under the *fixpoint* semantics where Connected is computed in layers, and new rows are added in each layer until no more row can be added.

```
Q9.
  with recursive Connected (origin, destination) as (
    select origin, destination
    from Flights
    union
    select c.origin, f.destination
    from Flights f, Connected c
    where c.destination=f.origin)
  select origin, destination
  from Connected;
```

Negation may show as a predicate (e.g., *name != 'Joe'*) in the where clause or the having clause or in the form of a set difference (e.g., Q3). Negation may also appear when a query uses the connector *not exists* (e.g., Q6). Recursion and negation do not work well together, and recursive SQL queries may not terminate with negation. For the same reason, SQL does not allow recursion using *not exists* and set difference.

The presence of *null* values adds one more level of complexity to SQL queries. For example, the predicate *origin='Seattle'* does not evaluate to true or false if origin of a flight is null. The semantics of predicates in SQL is defined under three-valued logic where expressions can evaluate to *true*, *false* and *unknown*.

3.3 Graph Model

The **graph model** describes entities as nodes and relationships as edges of a graph. Although the graph model has long been considered as an alternative to the relational model (e.g., see [178] and [72]) with queries expressed as graph patterns, it has only been recently placed on spotlight with the surging interest in knowledge graphs and RDF, a W3C standard for describing resources on the Web[2].

In a typical graph model for representing the knowledge of a domain using RDF, each entity is represented as a node, and the information about each entity is described using directed edges emanating from the node. An edge may connect an entity e_1 to another entity e_2, representing a relationship between the two entities or describing a predicate where e_1 is the subject and e_2 is the object (e.g., `AC 162 is operated by Air Canada`). An edge may also connect an entity to a literal node that has the value of an attribute for the entity. Figure 3.4 shows an example graph describing a flight, an airline and a few Canadian cities. Every entity and predicate has a unique id to make the statements more accurate. In our example, the ids are from Wikidata with their common prefixes not shown.

The graph model allows more complex relationships between entities without being constrained to fixed structure of tables. For example, to add a relationship between two entities e_1 and e_2, the underlying schema of the relational model must allow such relationships to be described whereas edges can be freely added between any arbitrary pair of nodes in the graph model.

3.3.1 SPARQL

SPARQL is the W3C query language for querying RDF where the data are seen as a ternary relation in the form of (*subject*, *predicate*, *object*), referred to as **triples** or **statements**. The language treats triples as first class citizens and allows each of subject, predicate and object in a triple pattern to be variables. It should be noted that RDF data may be queried using standard relational query languages such as SQL though the queries will not look as natural or intuitive as in SPARQL. Here is a SPARQL query to find all international airports in the US[3].

```
select ?airport
where
{
  ?airport wdt:P31 wd:Q644371.
```

[2] https://www.w3.org/RDF/

[3] Our SPARQL queries are posted on Wikidata and can be tried using the query service at `https://query.wikidata.org/`.

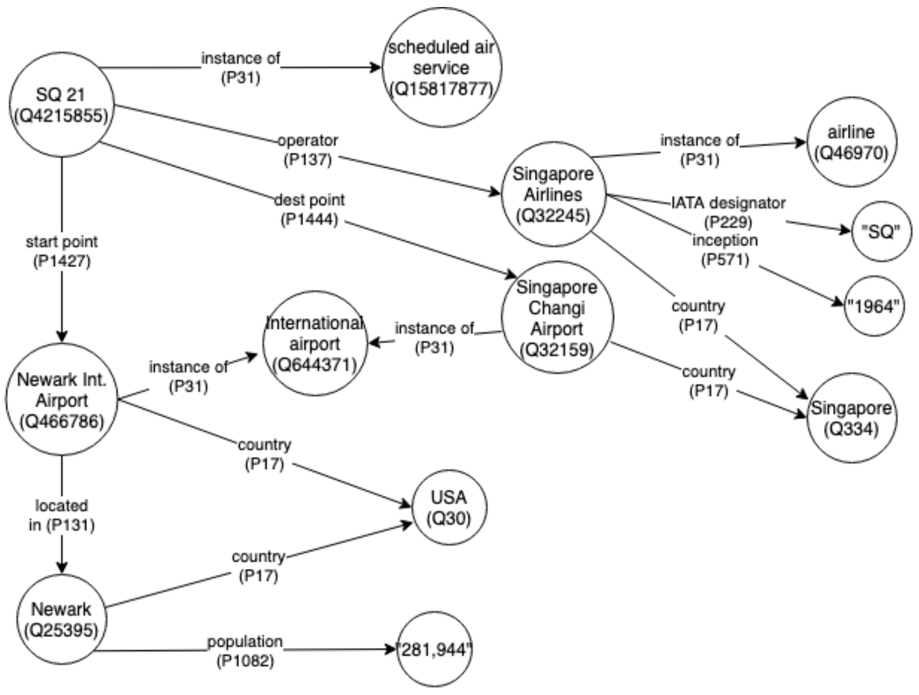


Fig. 3.4: An RDF example with entity and property labels from Wikidata

```
    ?airport wdt:P17 wd:Q30.
  }
```

Predicate and entity ids are from Wikidata (as shown in Figure 3.4). The first predicate binds ?airport to those entities that are instances of (wdt:P31) international airports (wd:Q644371) and the second predicate ensures the country (wdt:P17) of those airports is in the US (wd:Q30). The prefixes *wdt* and *wd* are predefined as follows:

```
  prefix wd: <http://www.wikidata.org/entity/>
  prefix wdt: <http://www.wikidata.org/prop/direct/>
```

Our next query finds top 10 big cities in the US that have the largest populations.

```
  select ?city
  where
  { ?city wdt:P31 wd:Q515.
    ?city wdt:P17 wd:Q30.
    ?city wdt:P1082 ?population.
    }
    order by desc(?population) limit 10
```

Similar to SQL, SPARQL allows aggregation and grouping. For example, our next query returns the number of international airports in each country. A grouping is done for each country and the results are ordered based on the number of international airports.

```
    select ?country (count(?airport) as ?c)
    where
    {
      ?airport wdt:P31 wd:Q644371.
      ?airport wdt:P17 ?country.
    }
    group by ?country
    order by desc(count(?airport))
```

SPARQL allows optional patterns to query edge types that may or may not be present and filter predicates to include and exclude certain graph patterns.

3.4 Storage and Indexing

A major strength of database query languages (including SQL and SPARQL) is their declarativeness: users describe the information to be stored and retrieved and the database system decides on the specifics of the storage structure and the evaluation algorithms. Each relational table may be stored as a heap (unsorted) or a sorted file, and indexes may be built to provide fast access to the data. Those indexes may be stored as separate files. Most commercial relational databases automatically build an index on the primary key of each table on the basis that the rows are often accessed using the primary key. Additional indexes can be built both by the user and the DBMS as needed. The choice of the storage structure under which the data is stored and the access paths the database engine takes in answering a query does not affect the query result but can have major implications in the running time.

As an example, consider our flight table given in Figure 3.3. The table may be organized as a heap file, with each disk page keeping a fixed number of records. Indices may be constructed on different columns to speed up the search. Figure 3.5 shows one such heap file with at most three rows stored in each page and a separate index file on the origin column. The index file, in our case, is sorted and provides an efficient access to records that match a given location of origin. The index file may be organized as a tree (e.g., B+ tree) or a hash for even greater efficiency of the accesses. Also more indexes may be constructed on other columns, such as destination and flight number, to allow more efficient searches on those columns. More details on file organization and indexing can be found in several textbooks [173, 266, 340].

ht

Edmonton	Page 1	r1
Denver	Page 2	r6
New York	Page 1	r3
San Jose	Page 3	r7
Seattle	Page 2	r4
Seattle	Page 3	r8
Toronto	Page 1	r2

(a) Index on origin

r1	AC	162	Edmonton	Toronto	7:30	2690	218	a1	
r2	AC	163	Toronto	Edmonton	8:30	2690	247	a2	Page 1
r3	UA	1274	New York	Seattle	6:45	3880	363	a3	
r4	UA	541	Seattle	Denver	9:03	1670	163	a3	
									Page 2
r6	UA	208	Denver	New York	13:30	2580	218	a4	
r7	UA	714	San Jose	Seattle	10:40	1129	140	a5	
r8	UA	1526	Seattle	San Jose	15:10	1129	125	a3	Page 3

(b) Flights as a heap file

Fig. 3.5: Flights table organized as a heap file with an index on origin

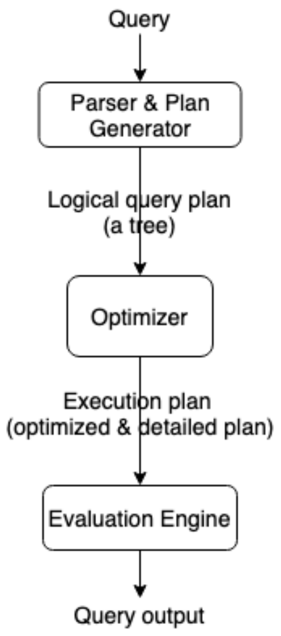

Query
Parser & Plan Generator
Logical query plan (a tree)
Optimizer
Execution plan (optimized & detailed plan)
Evaluation Engine
Query output

Fig. 3.6 Steps in query evaluation

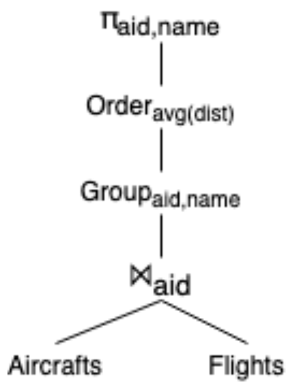
3.5 Query Evaluation and Optimization

As shown in Figure 3.6, query processing and optimization can be broken down into the following three steps: **query parsing and plan generation**, **query optimization** and **query evaluation**.

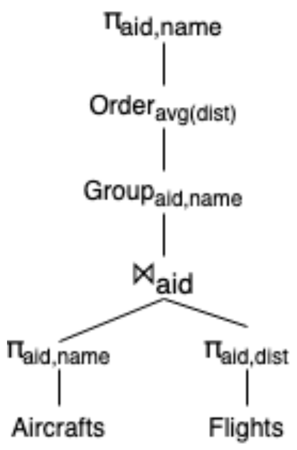
3.5.1 Query parsing and plan generation

During query parsing, the query syntax is first checked against the SQL grammar, and the syntactic pieces and categories (e.g., select list, from list, conditions) are identified. The parser may also enforce some semantic rules that are not in the grammar. For example, every relation referred in the query must exist in the schema, and every attribute mentioned in the `select` and `where` clauses must be an attribute of a relation in the current scope.

After the query passes those syntactic (and some non-syntactic) checks, the parse tree is mapped to a **logical query plan** where each leaf represents a base table in the query and each internal node represents a relational algebra operator over data, streaming through the children nodes. Each internal node in the tree pipes its result to the parent node, and the root node returns the result of the query. Figure 3.7a shows a query plan for one of our queries in Section 3.2. First, tables Aircrafts and Flights are joined based on aid, a common column between the two tables. The result is grouped based on aid and name and the groups are ordered before returning aid and name for each group.



(a) Query plan 1



(b) Query plan 2

Fig. 3.7: Two equivalent plans for Q3, the SQL query given in Section 3.2

3.5.2 Query optimizer

The same query semantics may be written in several different ways in SQL. Each SQL query may in turn be mapped to multiple different relational algebra expressions, each giving a query plan. The equivalence relationships between relational algebra expressions also allows a plan to be rewritten such that both the initial and rewritten plans are guaranteed to produce the same set of tuples on every legal database instance. Query optimizer is responsible for selecting the most efficient plan for a given query.

3.5.2.1 Query equivalence

Consider the two plans shown in Figure 3.7, which are equivalent with the projection distributing over the join, but the costs are expected to be different. In Query plan 2, the projection is pushed down in the tree and that should reduce the size of the intermediate results and, as a result, the overall execution cost. In general, the problem of detecting equivalence between two query expressions is *undecidable* [1] even if queries are limited to simple relational algebra. The problem becomes NP-hard if one limits the queries to a subset of relational algebra with only selection, projection and join, referred to as *conjunctive queries*. This will correspond to SQL queries with only *select*, *from* and *where* clauses and the predicates in the where clause limited to equality between columns and between columns and constants.

However, query optimizers avoid the decision problem by using equivalence rules to rewrite queries. Here are two examples of equivalence rules, showing that the join operation is both commutative and associative over relations $R1$, $R2$ and $R3$:

$$R1 \bowtie R2 \equiv R2 \bowtie R1,$$

$$(R1 \bowtie R2) \bowtie R3 \equiv R1 \bowtie (R2 \bowtie R3).$$

For a given query, one can construct many possible tress with the relations being queried at the leaves, and all these variations give rise to a large space of possible query plans each with a cost. For example, the number of full binary tress with n relations at the leaves is the well-known Catalan number $C(n-1) = \frac{(2n-2)!}{n!(n-1)!}$. The query optimizer has to go through all these plans to select a plan with the least cost. This means the optimizer has to estimate the cost of each query plan before it is evaluated. For large databases that reside on the secondary storage such as disk, the major cost is the number of I/Os.

3.5.2.2 Cost-based optimization

Cost estimation has been one of the contentious topics in database research, since an exact estimate often depends on many parameters that may not be known, such as the selectivity of a condition and the size of a join, or may change as tables are updated. With the result of each operation fed to the next operation in a query plan, estimating the size of intermediate results has a big impact on the overall cost estimates. Database systems maintain statistics of the tables and the columns to help in their size estimations. For example, maintaining the number of distinct values in a column can help with estimating the probability that a row is returned for an equality condition, assuming all values in the column are equally likely. When some values are more likely than others, a histogram and the set of top frequent values and their frequencies can provide a better estimate. The size of a join may be bounded from above by the size of the Cartesian product and from below by the size of a relation, for example when the join column is a key in one table.

A cost-based optimizer is expected to search the space of possible plans and find a query plan with the least cost. Join order selection is a common optimization type that benefits a large number of queries and is usually part of every query optimizer. Given a set of relations to be joined, different parenthesizations of those relations will give different join orders, and a query optimizer usually seeks a parenthesization with the least cost. Although the number of parenthesizations rises quickly with the number of relations, an efficient solution is using a dynamic programming algorithm that stores and reuses the results of computations. Adding other operations to a cost-based optimizer quickly increases the number of plans to a level that is hard to manage for large queries.

To reduce the size of the search space and the cost, some heuristics may be used on top of a cost-based optimizer. One heuristic that aims at reducing the size of the intermediate results is to perform selection and projection as early as possible. As another heuristic, the System R optimizer only considers join orders that form a left-deep tree, which provides more opportunities for pipelining. For example, with the right operand of a join stored in a table, there is only one input that is pipelined. Nested subqueries may be treated as functions that take some arguments as input and return a row or a set of rows as output. Some subqueries may be flattened and merged with the rest of the query using a join, whereas others may be cached for different values of input to avoid multiple evaluation of the same expressions. More information about query optimization in relational systems can be found in relevant literature [155, 47] and database textbooks [292].

3.5.3 Evaluation engine

indexquery evaluation Given a query plan (possibly obtained through a cost-based optimization), the evaluation engine may start from lowest level operations in the query plan and evaluate the query bottom up. A few strategies may be used to reduce the size of intermediate results and the cost of operations. One strategy is to evaluate an operation before attempting the next operation. Some operations such as sorting cannot produce any output before all their input is examined. The intermediate result of these **blocking operations** must be materialized or stored, whereas the output of non-blocking operations may be pipelined to the next operation in the evaluation order. Multiple operations may form a pipeline, where the result of one operation is fed to the next operation in the pipeline. Some advantages of pipelining include possible savings in the cost of writing and reading the intermediate data as well as generating early query results when the pipeline includes the root node of the query tree.

3.6 Summary

Some of the key concepts in databases are reviewed and discussed. In particular, the structure of data can be described using a conceptual model, and some of those models are reviewed. We have also reviewed SQL and relational algebra as two query languages for the relational model and SPARQL as a language for querying graph data stored or viewed as RDF. Given a query, there are a number of processes that the query has to go through for computing an answer. We have briefly reviewed some of those processes including query optimization and evaluation.

3.7 Further Reading

An introduction to the relational model and relational query languages such as Relational Algebra and SQL can be found in major database textbooks (for example, see [170, 292]). The same textbooks will also cover conceptual models, such as Entity-Relationship Model and UML, and the physical storage and indexing. For a deeper treatment of the topics including the foundations of relational databases and queries, query subclasses such as Conjunctive Queries, as well as the query equivalence, the readers can check out the classic book by Abiteboul et al. [1]. The initial system R optimizer was based on a dynamic programming algorithm for finding an optimal plan [278]. Modern query optimizers are more complex but may use different techniques including (but not limited to) dynamic programming to find an optimal plan among many candidates. See more recent surveys (e.g., [47, 124]) for a more indepth coverage of these techniques.