Chapter 7

Databases

What is the movie with the largest cast? How many directors have directed more than ten movies? The internet movie database (`www.imdb.com`) contains the answer to such questions, but writing a new program that evaluates the data in a specific way for every such question is laborious. Relational databases can store large amounts of structured data and answer possibly complex questions about it.

7.1 Relational Databases

Definition 7.1 (Table, Row, Column, Database)**.** *A **table** consists of **rows**, so that each row (data record) contains the same **fields**, i.e., kinds of entries. When the rows of a table are written line by line, the fields form the **columns** of the table. Each column is referred to by a descriptive name, and is associated with the type of the respective field, e.g., integer, floating point, string, or a date. A **database** is a collection of tables.*

Remarks:

- In the database context, tables are also called *relations*, because the entries in each row are related to each other, namely by belonging to the same row.

movies

title	director	year
12 Angry Men	Sidney Lumet	1957
Raiders of the Lost Ark	Steven Spielberg	1981
War of the Worlds	Steven Spielberg	2005
Manos: The Hands of Fate	Harold P. Warren	1966
⋮		

Figure 7.2: A database containing a single table called "movies" storing the title, director, and year of release for each movie.

Remarks:

- Databases as we study them are accessed using the so-called *structured query language* (SQL). Thus they are referred to as SQL or *relational* databases.
- There are also databases for storing other data, e.g., key-value pairs (Chapter 6), graphs, or whole documents. Such databases are sometimes called *NoSQL* databases.
- Some people pronounce SQL letter-by-letter, while others prefer to say "sequel", which stems from a predecessor with that name.
- MySQL and PostgreSQL are two popular open source SQL databases.
- MongoDB, CouchDB and Redis are popular open source NoSQL databases.
- Like http servers, SQL databases typically run as a daemon process on some server. Client applications connect to the server and authenticate themselves via username and password.
- Multiple users accessing the same database may result in concurrency issues. Some form of concurrency control is necessary!
- Other databases are tailored to single-user processing. They relieve developers from the burden of implementing efficient data structures for relational data. SQLite is one such example, and is used, e.g., in Firefox, Chrome, Android, Adobe Lightroom, and Windows 10.
- How can we store the data from Figure 7.2 using a SQL database?

7.2 SQL Basics

Remarks:

- All SQL statements end with a semicolon. The SQL language is case insensitive, but by convention keywords are often typed in upper case.
- The SQL specification is over 600 pages long. To add insult to injury there are lots of vendor specific "SQL dialects", i.e., modifications and extensions.
- However, the basic set of commands for creating, manipulating, and querying tables are largely the same across database implementations. The same is true for the basic data types.

Definition 7.3 (SQL Data Types)**.** *SQL defines the following types of columns.*

- CHARACTER(m) *and* CHARACTER VARYING(m) *for fixed and variable length strings of (maximum) length m,*
- BIT(m) *and* BIT VARYING(m) *for fixed and variable length bit strings of (maximum) length m,*
- NUMERIC, DECIMAL, INTEGER*, and* SMALLINT *for fixed point and integer numbers,*

- FLOAT, REAL, *and* DOUBLE PRECISION *for floating point numbers,*
- DATE, TIME, *and* TIMESTAMP *for points in time, and lastly*
- INTERVAL *for ranges of time.*

Remarks:

- The range of each type includes the special value NULL. Note that NULL is different from the string 'NULL', the empty string, and from the number 0 (zero). NULL indicates that the row has *no value* for the corresponding field.
- For the CHARACTER VARYING type, some database systems support strings of arbitrary length.
- Many databases implement more types, e.g., geographic coordinates, IP addresses, geometric objects, or large integers.

Listing 7.4 Creating the database moviedb containing a table movies.

```
1: CREATE DATABASE moviedb;
2: USE moviedb;
3: CREATE TABLE movies (
     title CHARACTER VARYING(200) NOT NULL,
     director CHARACTER VARYING(200) DEFAULT 'Steven Spielberg',
     year INTEGER
   );
```

CREATE DATABASE *database-name*;
Additional parameters allow to set database-specific options, e.g., user-based permissions, or default character sets for text strings. How a database is opened depends on the implementation. Listing 7.4 shows how to do it in MySQL.

CREATE TABLE *table-name* (*field-name type, field-name type, . . .*);
To enforce that all rows have a value for a particular field, one can add NOT NULL to the type when creating the table. Fields have a default value, which is NULL if not specified by adding DEFAULT *value* to the type description.

Remarks:

- There are also GUI and web-based client applications (that execute locally or on an http-server, respectively) and offer access to the database in a more intuitive manner than the classic command line tools. Examples for web-based interfaces are phpPgAdmin and phpMyAdmin for PostgreSQL and MySQL, respectively.
- Such tools are especially helpful for creating the databases and tables. They also feature importing data from various formats, e.g., CSV files, instead of using SQL statements to populate the tables.

Listing 7.5 Populating the movies table with data.

```
4: INSERT INTO movies
     (title, director, year) VALUES
     ('12 Angry Men', 'Sidney Lumet', 1957),
     ('Raiders of the Lost Ark', DEFAULT, 1981),
     ('War of the Worlds', DEFAULT, 2005),
     :
     ('Manos: The Hand of Fate', 'Harold P. Warren', 1966)
   ;
```

INSERT INTO *table-name* (*field-name*, . . .) VALUES (*value*, . . .);
Values must be listed in the same order as the corresponding field names. When a field name (and thus its value) is omitted the field's default value is assumed. When the list of field names is omitted the field's values must be listed in the same order that was used when creating the table. To insert more than one row in one statement, multiple rows may be separated by a comma.

Listing 7.6 Querying the movies table.

```
5: SELECT * FROM movies;
6: SELECT * FROM movies WHERE director = 'Steven Spielberg';
7: SELECT title FROM movies WHERE year BETWEEN 1990 AND 1999;
8: SELECT * FROM movies WHERE title IS NULL OR director IS NULL;
9: SELECT title, director FROM movies WHERE title LIKE '%the%';
```

SELECT *field-name*, . . . FROM *table-name* WHERE *condition*;
Lists all specified fields of all rows in the table that fulfill the condition. The special field * lists all fields. The WHERE condition may be omitted to list the whole table. A condition can include comparisons ($<, >, =, <>$) between fields constants. The special value NULL can be tested with IS NULL. Conditions can be joined using parenthesis and logic operators like AND, OR, and NOT. Strings can be matched with patterns using ***field-name* LIKE *pattern***. In the pattern, an underscore (_) matches a single character, whereas % matches arbitrarily many.

Listing 7.7 Aggregation with SQL.

```
10: SELECT MIN(year) FROM movies;
11: SELECT AVG(year) FROM movies WHERE director='Sidney Lumet';
12: SELECT COUNT(*) FROM movies;
13: SELECT COUNT(DISTINCT director) FROM movies;
```

SELECT *aggregate*, . . . ;
Functions for aggregation include AVG to compute the average of a certain field, MIN and MAX for the minimum and maximum value, SUM for the sum of a field, and COUNT to count the number of occurrences. In an aggregation, the keyword DISTINCT indicates that only distinct values should be considered.

Remarks:

- Query 12 in Listing 7.7 returns the number of entries in the table, whereas query 13 returns the number of different movie directors.

Listing 7.8 Grouping and sorting.

```
14: SELECT director, COUNT(title) FROM movies GROUP BY director;
15: SELECT director, COUNT(title) FROM movies GROUP BY director
    HAVING COUNT(title)>10;
16: SELECT year, director, COUNT(title) FROM movies
      GROUP BY director, year
      ORDER BY year DESC, director ASC;
```

SELECT *field-name*|*aggregate*, ... GROUP BY *field-name*,...;
Aggregations may be partitioned using the group-by clause. Similar to before, the query result can only include aggregates and fields by which the result is partitioned.

Since WHERE clauses are applied before GROUP BY the result of aggregations cannot appear in them. When the result should be conditioned on the result of an aggregation, a HAVING clause can be used.

Remarks:

- Query 14 in Listing 7.8 reports how many movies of each director are in the database, and query 16 breaks the same down by year.

SELECT ... ORDER BY *field-name*,...;
After each field-name, the keyword ASC or DESC can be used to determine ascending or descending sorting order, respectively.

Listing 7.9 Updating and removing rows.

```
17: UPDATE movies SET title = 'Star Wars Episode IV: A New Hope'
      WHERE title = 'Star Wars';
18: DELETE FROM movies WHERE title = '';
```

UPDATE *table* SET *field-name* = *value*,... WHERE *condition*;
Updates the specified fields in all rows fulfilling the condition.

DELETE FROM *table-name* WHERE *condition*;
Removes all rows fulfilling the condition from the table.

7.3 Modeling

The way our example table from Figure 7.2 is designed results in lots of duplicate data—the director's name is stored anew for each row, and two directors with the same name cannot be distinguished. The situation worsens when we want to store the cast of each movie. Clearly the way we modeled our data can be improved. *Entity-Relationship* (ER) diagrams are a tool to find good representations for data.

Definition 7.10 (ER Diagram)**.** *Rectangles denote* ***entities*** *(tables), and diamonds with edges to entities indicate* ***relations*** *between those entities. On such an edge, the number* 1 *or the letter* n *denotes whether the corresponding entity takes part once or arbitrarily many times in the relation. Entities and relations can have* ***attributes*** *(columns) with a name, drawn as ellipses. Italicised attributes are* ***key attributes*** *which must be unique for each such entity.*

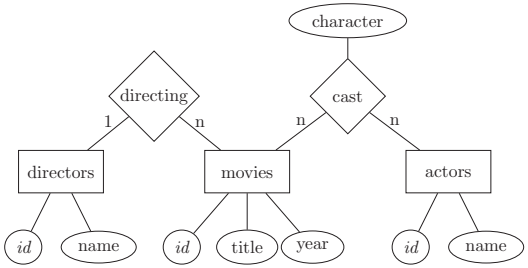

Figure 7.11: Model for a movie database. Movies and directors are in a 1-to-n relation: Each movie is directed by 1 director, and a director may work on many movies. Movies and actors are in a n-to-n relation, which has an additional attribute: An actor may appear in many movies, and each appearance is associated with a character in that movie, played by that actor.

Remarks:

- It is standard practice to assign a so-called *key attribute*, often named *id*, to every entity.
- What do ER diagrams have to do with SQL? Primarily, ER diagrams are for conceptually modeling the kind of data and relations one wishes to store. They can be translated into databases, but not in a unique way.
- A close relative of the ER diagram is the Unified Modeling Language (UML). UML is used to represent the tables of a database (or classes of object oriented software) accurately, with detailed information, e.g. fields.
- Each entity corresponds to a table with the corresponding attributes as columns. An n-to-n relation is represented by a table with columns for each attribute, and a column for the key attribute of each entity in the relation.

actor

id	name
1	Harrison Ford
2	Tom Cruise
⋮	

cast

actor_id	character	movie_id
1	Indy	2
2	Ray Ferrier	3
⋮		

Figure 7.12: The actor table and a table capturing the cast relation.

Remarks:

- The same scheme can be used for 1-to-1 and 1-to-n relations. However, one may also include the relation in the table storing the entity on the 1-side.

directors

id	name
1	Sidney Lumet
2	Steven Spielberg
3	Harold P. Warren
⋮	

movies

id	title	year	director_id
1	12 Angry Men	1957	1
2	Raiders of the Lost Ark	1981	2
3	War of the Worlds	2005	2
4	Manos: The Hands of Fate	1966	3
⋮			

Figure 7.13: The movie and director tables using the new database layout. The director table simply maps ids to director names. Since the directing relationship is 1-to-n, it can be represented by adding a column to the movies table that stores the director for each movie.

Remarks:

- Similarly, a 1-to-1 relation can be turned into an attribute of one of the entities.
- Tables dedicated to capturing relations are often called *join* tables.

7.4 Joins

How can we access the data, which is now scattered across multiple tables?

Listing 7.14 A query that returns the table presented in Figure 7.13.

```
1: SELECT movie.title, director.name AS director, movie.year FROM movie
     INNER JOIN director ON movie.director_id = director.id;
```

SELECT ...
FROM ***left-table*** **INNER JOIN** ***right-table*** **ON** ***condition*****;**
Returns all rows that can be formed from a row in the left-table and a row in the right-table that satisfy the specified condition.

Remarks:

- In a query, one can create aliases for field and table names using the AS keyword, see Listing 7.14.
- A row in one of the tables that does not have a matching row (that satisfies the condition) in the other table will not appear in the result. For example, a director with id 5 would not appear, since there are no movies that reference that director_id.
- An INNER JOIN where the condition is TRUE returns the carthesian product of both tables. This special case can also be obtained with the CROSS JOIN clause.
- OUTER JOINs include also unmatched rows.

SELECT ...
FROM ***left-table*** **LEFT|RIGHT|FULL OUTER JOIN** ***right-table*** **ON** ***condition*****;**
Returns all rows from the inner join. In addition, a LEFT or RIGHT OUTER JOIN also returns all rows from the left or right table that have no matching row on the opposite table, respectively. The fields in unmatched rows that cannot be filled from the other table are filled with NULL values. A FULL OUTER JOIN returns both of the above.

Remarks:

- A RIGHT OUTER JOIN lists the movies that have a director and include all "directors" that have not directed any movie. A LEFT OUTER JOIN includes the movies with no director instead.
- The result of a JOIN clause can be ordered, fields can be aggregated and grouped, and conditions can be added using WHERE clauses.
- We can combine joins and aggregations to answer our initial question:

Listing 7.15 Finding the 10 movies with the largest cast.

```
SELECT movie.title, COUNT(*) AS cast_size
FROM cast RIGHT OUTER JOIN movie ON cast.movie_id = movie.id
GROUP BY movie.id ORDER BY cast_size DESC LIMIT 10;
```

Remarks:

- The query from Listing 7.15 uses a LIMIT clause to return only the ten first entries of the sorted results.
- We used a RIGHT OUTER JOIN to make sure also movies without a cast are taken into account.
- Queries may use more than one JOIN clause.

Listing 7.16 Finding all movies that Harrison Ford did not appear in.

```
3: SELECT movie.title
   FROM actor INNER JOIN cast
   ON cast.actor_id = actor.id AND actor.name = 'Harrison Ford'
   RIGHT OUTER JOIN movie ON cast.movie_id = movie.id
   WHERE cast.actor_id IS NULL;
```

Remarks:

- The conditions for the first join in Listing 7.16 ensure that only movies with Harrison Ford are taken into account for the second OUTER JOIN. That second join in turn delivers all movies that cannot be matched, yielding a NULL entry for the actor_id for movies without Harrison Ford.
- To ensure that every row in the cast table contains a value one can specify that the fields are NOT NULL when creating the table.

7.5 Keys & Constraints

What is stopping us from inserting a row in the cast table that contains an actor_id or a movie_id that does not exist? Or from creating a director with a duplicate id?

Definition 7.17 (Key)**.** *In a table, a column (or set of columns) is a* ***unique key*** *if the corresponding values uniquely identify the rows within the table. The* ***primary key*** *of a table is a designated unique key. A* ***foreign key*** *is a column (or set of columns) that references the primary key of another table.*

Remarks:

- SQL databases can automatically enforce these constraints. For example, a row containing a foreign key can only be inserted if it references an existing primary key. Vice versa, a row may only be removed if its primary key is not referenced by any foreign key.

Listing 7.18 Adding constraints to the database.

```
1: ALTER TABLE movies ADD CONSTRAINT UNIQUE (actor_id, character, movie_id);
2: ALTER TABLE director ADD PRIMARY KEY id;
3: ALTER TABLE movies
     ADD FOREIGN KEY (director_id) REFERENCES director;
```

ALTER TABLE *table*
ADD CONSTRAINT UNIQUE (*field-name*,...);
The values held by the specified fields must be unique among all rows.

ALTER TABLE *table* ADD PRIMARY KEY (*field-name*,...);
Sets the specified fields as the primary key for the table. Doing so also ensures that no duplicate entry is present when inserting or updating data.

ALTER TABLE *left-table* ADD FOREIGN KEY (*field-name*,...)
REFERENCES right-table;
Ensures that the values in the specified fields in the left table are the primary key of a row in the right table.

Remarks:

- Constraints for new tables can also be set using CREATE TABLE.
- Other ALTER TABLE queries add different constraints (e.g., checking that an integer field contains only certain values), remove constraints, and change the name, type or default value of fields.
- To ensure that checking constraints and searching for data is fast, databases rely on *index* data structures.

7.6 Indexing

Definition 7.19 (Index)**.** *In the database context, an* ***index*** *is a data structure that speeds up searching for rows with specific values.*

Remarks:

- Without an index data structure, rows with a specific value can only be found by scanning through the whole table.
- In Chapter 6 you learned that hash tables can retrieve the row associated with a key in expected constant time. Many databases implement hash tables as one possible index data structure.

Listing 7.20 Adding a hash table index to our database.

```
1: CREATE INDEX directorid ON director USING HASH (id);
```

Remarks:

- The director associated with a movie is now found quickly when performing a join.
- Some database implementations automatically create index data structures to speed up queries that involve frequently used fields.
- Index data structures have a name—"directorid" in Listing 7.20. This is for referencing it later, e.g., if one decides to delete the index.
- Hash tables scatter the data across the storage (volatile or persistent), and it is likely that every access incurs overhead. Many database queries require scanning through ranges of the data sequentially. For example, when searching the movies from 2000–2005. Thus, accessing supposedly closeby rows requires accessing items at many different places.

- B+ trees are a data structure designed to minimize the amount of I/O operations for both searching and scanning.

Listing 7.21 Adding a B+ tree index to our database.

```
1: CREATE INDEX movieyear ON movies USING BTREE (year);
```

Definition 7.22 (B+ Tree)**.** *A **B+ Tree** of order b is a rooted search tree mapping **keys** to **rows**. In a B+ Tree, every non-leaf node has between $\lfloor b/2 \rfloor$ and b children, whereas leaf nodes have between $\lfloor (b-1)/2 \rfloor$ and $b-1$ children. A non-leaf node v with i children contains exactly $i-1$ keys, in sorted order. The keys contained in the sub-tree rooted at v's i^{th} child are between the $i-1^{st}$ and i^{th} key contained in node v.*

*B+ trees are **balanced**, i.e., all leaf nodes are at the same depth. Leaf nodes contain all keys inserted into the tree, and the child pointer corresponding to key k is used to point to the row associated with k. The unused child pointer of a leaf w is used to point to w's next sibling.*

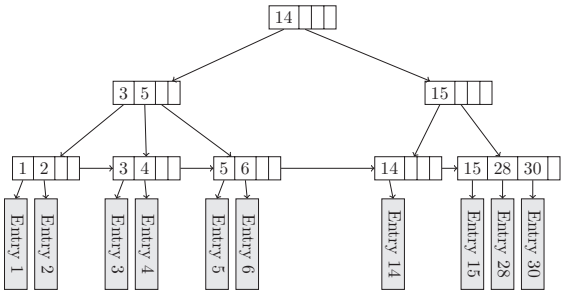

Figure 7.23: Example B+ tree of order $b = 4$.

Remarks:

- The root node is a special case—it may have as little as 1 child if it is a leaf itself, or 2 children if it is an inner node.
- The order b is sometimes called *branching factor*. To reduce the number of necessary I/O operations, b is chosen so that all data necessary to store a node is the size of (at least) one block on the disk/one cache line.
- Finding the row for some key k in a B+ tree works similar to a binary search tree.
- When inserting a key k it may happen that the leaf v that should contain k is already full. In that case v, and possibly predecessors of v that contain too many keys, need to be split.

Algorithm 7.24 B+SplitUp(k, r)

```
1: Given a B+ tree, a key k, and a node v
2: Create a new node v'
3: Distribute k and the keys in v among v and v' s.t. v' gets the larger keys
   and both nodes are half filled
4: Let k' be the smallest key in v'
5: Let p be v's parent
6: if p is full then
7:    SplitUp(p, k')
8: end if
9: Insert k with child v at node p
```

Remarks:

- If the root node is split into two nodes v, v', then a new root r containing key k and v and v' as children is created, and the recursion stops.
- Inserting a key k is now performed by first making room using B+SplitUp if necessary, and then inserting k at the leaf.

Algorithm 7.25 B+Insert(k, r)

```
1: Given a B+ tree, a key k, and a row r
2: Perform a search for k to find the leaf v at which k must be inserted
3: if v contains b − 1 keys then
4:    B+SplitUp(v, k)
5:    Replace child of key k with row r in node v
6: else
7:    Insert key k with row r into node v
8: end if
```

Remarks:

- Vice versa, when deleting a key, nodes with too few keys need to be filled up or removed from the tree.

Algorithm 7.26 B+MergeUp(v)

```
1: Given a node v containing less than (b − 1)/2 keys
2: Let l and r be the left and right sibling of v
3: if l contains more than (b − 1)/2 keys then
4:    Move largest key x from v's left sibling to v
5:    Update key in parent corresponding to v to x
6: else if r has more than (b − 1)/2 keys then
7:    Move smallest key x from r to v
8:    Update key in parent corresponding to r to x
9: else
10:   Merge all keys of v and one of v's siblings (use the node that is further to
      the left to store the keys)
11:   Remove the now empty node and its corrpesponding key from v's parent
12:   if v's parent contains less than (b − 1)/2 keys then
13:      B+MergeUp(p)
14:   end if
15: end if
```

Remarks:

- If v does not have a left or right sibling, the corresponding **if**-statement is ignored.
- The properties of B+ trees ensure that every node has at least one sibling. Thus, the merge operation (Lines 10–14) always has two nodes to work with.
- If no keys can be "borrowed" from a sibling, the merge may propagate until the last two children of the root node are merged into one node. In that case the root node is replaced by the merged node, decreasing the height of the tree by 1.

Algorithm 7.27 B+Delete(k)

```
1: Given a B+ tree and a key k
2: Perform a search for k to find the leaf v containing k
3: Remove k from v
4: if v contains less than (b − 1)/2 keys then
5:    B+MergeUp(v)
6: end if
```


Remarks:

- The height of a B+ tree is changed only when inserting a new or removing an old root node. Therefore, all leaf nodes are always at the same depth, thus ensuring the *balanced* property.
- A B+ tree containing n keys has height at most $O(\log_b n)$.
- It may happen that many nodes contain as little as $b/2$ keys, wasting memory and I/O operations. B* trees ensure that nodes contain at least $2/3b$ keys by cleverly "trading" entries with neighboring nodes when they contain too many or too few keys.

7.7 Transactions

Definition 7.28 (Transaction)**.** *A database* ***transaction*** *is a sequence of statements that is executed atomically.*

Remarks:

- Why would we need transactions? Consider a bank managing customer's accounts using a database system. Alice wants to calculate the liquid assets, and Bob wants to make a money transfer:

Listing 7.29 Concurrency issues in databases.

```
   Alice's statement:
1: SELECT SUM(balance) FROM accounts;

   Bob's statements:
2: UPDATE accounts SET balance=balance−100 WHERE customer='Bob';
3: UPDATE accounts SET balance=balance+100 WHERE customer='Jim';
```

Remarks:

- Assuming that the database uses multiple threads or processes to process queries, Alice's query may be CHF 100 short.
- To execute the queries atomically, both Alice and Bob can use transactions.

BEGIN TRANSACTION; ***statement*$_1$**; ...; END TRANSACTION;**
Executes the statements atomically.

Remarks:

- One way to implement transactions is to keep track of all fields read from and written to (the read- and write-set, respectively). Then, before a transaction ends, the database system checks whether another transaction wrote to any value in the read-set. If the read-set is unchanged, the write-set can be applied atomically, e.g., by using a global lock.

- SQL offers different so-called isolation levels. The isolation level defines when writes of one transaction become visible to others. The above technique implements the *repeatable reads* level, ensuring that read values were committed before and are not written by another transaction.
- Consider some transaction A that selects all years between 1999 and 2004. What happens if another transaction B concurrently inserts an entry for the year 2000? In the *repeatable reads* isolation level, A may not see B's data if B's insert is scheduled *after* A read all other entries for the year 2000, and A would still be allowed to finish. *Repeatable reads* do not ensure atomicity ...
- The highest isolation level is called *serializable*. This level ensures that the transactions behave "as if they were executed in some sequential order", possibly at the cost of low concurrency.

7.8 Programming with Databases

How do you write an application that relies on a SQL database to store data? Should you construct the necessary SQL statements by manipulating strings, send them to the SQL server, and then parse the result?

Remarks:

- Writing such a SQL client is one possibility, but this is error-prone: The compiler used for the application will not be able to detect errors made in the SQL statements. Moreover, the declarative SQL most likely does not mix well with the programming language chosen for the application.
- One way to mitigate these issues in object oriented programming languages is object/relational mapping.

Definition 7.30 (Object/Relational Mapping)**.** ***Object/Relational Mapping (ORM)*** *is a design pattern used in object oriented programming to store objects in and retrieve them from relational (SQL) databases.*

Remarks:

- In the simplest case, an ORM simply maps a class to a table. An object then corresponds to a row, and the object's attributes correspond to the row's fields.
- The ORM takes care of storing and retrieving object in the database and performs type conversions where necessary. It provides object oriented abstractions for database queries involving WHERE and other clauses. ORMs also remove boilerplate code, i.e., setting up the SQL connection, error handling, data conversion, etc.
- This way no—or only very little—SQL code "leaks" into the object-oriented program.

- Popular ORMs include SQLAlchemy for Python, ActiveRecord for Ruby, Hibernate for Java, and the Entity Framework for .NET.

Listing 7.31 Using Hibernate for Java to change the personal information of an existing director.

```
1: Director director = session.load(Director.class,new Long(3464377));
2: // director: id = 3464377, name = "Larry Wachowski", gender = "m"
3: director.setName('Lana Wachowski');
4: director.setGender('f');
5: commit();
```

Remarks:

- The ORM needs to know how it should translate between objects and rows. For that, many ORM implementations allow to specify the database layout using object oriented methods.
- Many ORM mappers also support creating the database using the object oriented specification. This ensures that the database and what the ORM expects are kept in sync.
- What if you need to add or remove a column without deleting and re-creating the database? There are so-called migration tools that facilitate this process.
- Some concepts from object oriented programming are difficult to model with database concepts, and vice versa. The problems arising from combining these two paradigms are called the *Object-relational impedance mismatch*.

Chapter Notes

In 1970, Edgar F. Codd proposed the relationad database model [5] while working at IBM research. Later in the 70s, another group at IBM developed SQL's predecessor SEQUEL (Structured English QUEry Language) [3]. After being renamed SQL due to trademark issues, it was standardized by the ISO in 1987 and later revised [7]. Other companies started developing relational databases, and nowadays there are many SQL databases implementing different feature sets to choose from.

Around the same time, ER diagrams were conceived as a modeling tool [2, 4]. The Unified Modeling Language (UML), first standardized by the ISO in 1995 [8] and revised in 2012, also includes diagrams that model databases.

B Trees were invented in 1970 [1] for use in file systems. Many variants were studied, among them B* Trees [9], in which at most 1/3 of the memory is unused instead of 1/2 for B Trees. People soon realized that (also for file systems) scanning subsequent rows is an important operation. B+ Trees require at most one I/O operation to find the next element, cf. [9, 6].

Techniques from database systems can also be found in other areas of computer science. Transactions as a parallel programming model have been adotped

for other programming languages under the term *transactional memory*. Ideas developed to ensure that database transactions appear atomic w.r.t. writing data to disk were adopted by general purpose file systems under the name *journaling*.

This chapter was written in collaboration with Jochen Seidel.

Bibliography

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, 1970.

[2] A. P. G. Brown. Modelling a real world system and designing a schema to represent it. In *IFIP TC-2 Special Working Conference on Data Base Description*, 1975.

[3] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74. ACM, 1974.

[4] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1976.

[5] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 1970.

[6] Douglas Comer. The ubiquitous B-Tree. *ACM Comput. Surv.*, 1979.

[7] International Organization for Standardization. Information technology – Database languages – SQL – part 1: Framework (SQL/Framework), 2011. ISO/IEC 9075-1.

[8] International Organization for Standardization. Information technology – Object Management Group Unified Modeling Language (OMG UML) – Part 1: Infrastructure, 2012. ISO/IEC 19505-1.

[9] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1973.