Community Experience Distilled

Learning PostgreSQL

Create, develop, and manage relational databases in real-world applications using PostgreSQL

Salahaldin Juba Achim Vannahme
Andrey Volkov

[PACKT] PUBLISHING open source* community experience distilled

Learning PostgreSQL

Create, develop, and manage relational databases in real-world applications using PostgreSQL

Salahaldin Juba

Achim Vannahme

Andrey Volkov

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Learning PostgreSQL

First published: November 2015

Production reference: 1241115

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-918-8

www.packtpub.com

Credits

Authors
Salahaldin Juba
Achim Vannahme
Andrey Volkov

Reviewers
Ângelo Marcos Rigo
Dr. Isabel Rosa

Commissioning Editor
Julian Ursell

Acquisition Editors
Tushar Gupta
Greg Wild

Content Development Editor
Parita Khedekar

Technical Editor
Vijin Boricha

Copy Editors
Shruti Iyer
Sonia Mathur

Project Coordinator
Judie Jose

Proofreader
Safis Editing

Indexer
Monica Ajmera Mehta

Graphics
Disha Haria

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

About the Authors

Salahaldin Juba has over 10 years of experience in industry and academia, with a focus on database development for large-scale and enterprise applications. He holds a master's degree of science in environmental management and a bachelor's degree of engineering in computer systems.

I would like to express my deepest gratitude to my colleagues Achim Vannahme and Andrey Volkov for making this work possible. Also, I would like to thank all those who provided support, especially the Packt publishing team, especially the persons whom I interacted with—Vijin Boricha, Greg Wild, and Parita Khedekar—for their great help with proofreading, design, comments, and remarks.

I would also like to thank my mother, Wedad; my wife, Rana; and the rest of my family, who supported me despite all of the time that I had to devote to this book over them.

Achim Vannahme works as a senior software developer at a mobile messaging operator, where he focuses on software quality and test automation. He holds a degree in computer science and has over 10 years of experience in using Java and PostgreSQL in distributed and high-performance applications.

Andrey Volkov pursued his education in information systems in the banking sector. He started his career as a financial analyst in a commercial bank. Here, Andrey worked with a database as a data source for his analysis and soon realized that querying the database directly is much more efficient for ad hoc analyses than using any visual report-generating software. He joined the data warehouse team, and after a while, he led the team by taking up the position of a data warehouse architect. Andrey worked mainly with Oracle databases to develop logical and physical models of finance and accounting data, created them in a database, implemented procedures to load and process data, and performed analytical tasks. He was also responsible for teaching users how to use data warehouse and BI tools, and SQL training was a part of his job as well.

After many years of being greatly interested in the aspects of his job that were related to IT rather than accounting or banking, Andrey changed fields. Currently, he works as a database developer in a telecommunication company. Here, Andrey works mainly with PostgreSQL databases and is responsible for data modeling, implementing data structures in databases, developing stored procedures, integrating databases with other software components, and developing a data warehouse.

Having worked with both Oracle and PostgreSQL—the former is a leading commercial and the latter is one of the most advanced open source RDBMSes—he is able to compare them and recognize and evaluate the key advantages of both. Andrey's extensive experience, therefore, made him able and willing to work on this book.

About the Reviewers

Ângelo Marcos Rigo has a strong background in web development, which he has worked with since 1998, with a focus on content management systems, hibryd mobile apps and custom web based systems. He holds a degree in systems information and also has extensive experience in managing, customizing, and developing extensions for the moodle LMS. Ângelo can be reached on his website, `http://www.u4w.com.br`, for consultation. He has also reviewed, *Moodle Security, Packt Publishing*.

I would like to thank my wife, Janaina de Souza, and my daughter, Lorena Rigo, for their support while I was away to review this book.

Dr. Isabel Rosa is a research associate at Imperial College London and one of the cofounders of Earthindicators. She has a PhD in computational ecology from Imperial College London and extensive experience in data mining and predictive modeling. For the last five years, Dr. Rosa worked as a researcher with Imperial College London. During her academic career, she acquired several skills such as statistical analysis, programming (R, C++, Python), working with geographic information systems (ArcGIS and QGIS), and creating databases (PostgreSQL/PostGIS, SQLServer). Dr. Rosa is also the lead author and coauthor of several scientific papers published in top-quality scientific journals, such as Global Change Biology. She has presented her work at several national and international scientific conferences and is the lead coordinator of Land Use Forum (London).

Table of Contents

Preface

Picking the right database management system is a difficult task due to the vast number of options on the market. Depending on the business model, one can pick a commercial database or an open source database with commercial support. In addition to this, there are several technical and nontechnical factors to assess. When it comes to a relational database management system, PostgreSQL stands at the top for several reasons. The PostgreSQL slogan, "The world's most advanced open source database", shows the sophistication of PostgreSQL features and community confidence.

PostgreSQL is an open source object relational database management system. It emphasizes extensibility and competes with major relational database vendors such as Oracle, SQL server, and MySQL. Due to its rich extensions and open source license, it is often used for research purposes, but PostgreSQL code is also the base for many commercial database management systems such as Greenplum and Vertica. Furthermore, start-up companies often favor PostgreSQL due to its licensing costs and because there are a lot of companies that provide commercial support.

PostgreSQL runs on most modern operating systems, including Windows, Mac, and Linux flavors. Also, there are several extensions to access, manage, and monitor PostgreSQL clusters, such as pgAdmin III. PostgreSQL installation and configuration is moderately easy as it is supported by most packaging tools, such as yum and apt.

Database developers can easily learn and use PostgreSQL because it complies with ANSI SQL standards and comes with many client tools such as psql and pgAdmin III. Other than this, there are a lot of resources to help developers learn PostgreSQL; it has a very good documentation manual and a very active and organized community.

PostgreSQL can be used for both OLTP and OLAP applications. As it is ACID compliant, it can be used out of the box for OLTP applications. For OLAP applications, PostgreSQL supports Window functions, FDW, and table inheritance; there are many external extensions for this purpose as well.

Even though PostgreSQL is ACID compliant, it has very good performance as it utilizes state of the art algorithms and techniques. For example, PostgreSQL utilizes MVCC architecture to allow concurrent access to data. Also, PostgreSQL provides a very good analyzer and advanced features, such as data partitioning using table inheritance and constraint exclusion, to speed up the handling of very large data. PostgreSQL supports several types of indexes such as B-Tree, GiN, and GiST, and BRIN indexes are also supported by PostgreSQL 9.5 at the time of writing this book.

PostgreSQL is scalable thanks to the many replication solutions in the market, such as Slony and pgpool-II. Additionally, PostgreSQL supports out-of-the-box synchronous and asynchronous streaming replication. This makes PostgreSQL very attractive because it can be used to set up highly available and performant systems.

What this book covers

Chapter 1, *Relational Databases*, introduces relational database system concepts, including relational database properties, relational algebra, and database modeling. Also, it describes different database management systems such as graph, document, key value, and columnar databases.

Chapter 2, *PostgreSQL in Action*, provides first-hand experience in installing the PostgreSQL server and client tools on different platforms. This chapter also introduces PostgreSQL capabilities, such as out-of-the-box replication support and its very rich data types.

Chapter 3, *PostgreSQL Basic Building Blocks*, provides some coding best practices, such as coding conventions, identifier names, and so on. This chapter describes the PostgreSQL basic building blocks and the interaction between these blocks, mainly template databases, user databases, tablespaces, roles, and settings. Also, it describes basic data types and tables.

Chapter 4, *PostgreSQL Advanced Building Blocks*, introduces several building blocks, including views, indexes, functions, user-defined data types, triggers, and rules. This chapter provides use cases of these building blocks and compares building blocks that can be used for the same case, such as rules and triggers.

Chapter 5, *SQL Language*, introduces Structured Query Language (SQL) which is used to interact with a database, create and maintain data structures, and enter data into databases, change it, retrieve it, and delete it. SQL has commands related to Data Definition Language (DDL), Data Manipulation Language (DML), and Data Control Language (DCL). Four SQL statements form the basis of DML—SELECT, INSERT, UPDATE, and DELETE—which are described in this chapter.

The SELECT statement is examined in detail to explain SQL concepts such as grouping and filtering to show what SQL expressions and conditions are and how to use subqueries. Some relational algebra topics are also covered in application to joining tables.

Chapter 6, *Advanced Query Writing*, describes advanced SQL concepts and features, such as common table expressions and window functions. This helps you implement a logic that would not be possible without them, such as recursive queries. Other techniques explained here, such as the DISTINCT ON clause, the FILTER clause, or lateral subqueries, are not that irreplaceable. However, they can help make a query smaller, easier, and faster.

Chapter 7, *Server-Side Programming with PL/pgSQL*, describes PL/pgSQL. It introduces function parameters, such as the number of returned rows, and function cost, which is mainly used by the query planner. Also, it presents control statements such as conditional and iteration ones. Finally, it explains the concept of dynamic SQL and some recommended practices when using dynamic SQL.

Chapter 8, *PostgreSQL Security*, discusses the concepts of authentication and authorization. It describes PostgreSQL authentication methods and explains the structure of a PostgreSQL host-based authentication configuration file. It also discusses the permissions that can be granted to database building objects such as schemas, tables, views, indexes, and columns. Finally, it shows how sensitive data, such as passwords, can be protected using different techniques, including one-way and two-way encryption.

Chapter 9, *The PostgreSQL System Catalog and System Administration Functions*, provides several recipes to maintain a database cluster, including cleaning up data, maintaining user processes, cleaning up indexes and unused databases objects, discovering and adding indexes to foreign keys, and so on.

Chapter 10, *Optimizing Database Performance*, discusses several approaches to optimize performance. It presents PostgreSQL cluster configuration settings, which are used in tuning the whole cluster's performance. Also, it presents common mistakes in writing queries and discusses several approaches to increase performance, such as using indexes or table partitioning and constraint exclusion.

Chapter 11, *Beyond Conventional Data types*, discusses several rich data types, including arrays, hash stores, and documents. It presents use cases as well as operations and functions for each data type. Additionally, it presents full-text search.

Chapter 12, *Testing*, covers some aspects of the software testing process and how it can be applied to databases. Unit tests for databases can be written as SQL scripts or stored functions in a database. There are several frameworks that help us write unit tests and process the results of testing.

Chapter 13, *PostgreSQL JDBC*, introduces the JDBC API. It covers basic operations, including executing SQL statements and accessing their results as well as more advanced features such as executing stored procedures and accessing the metainformation of databases and tables.

Chapter 14, *PostgreSQL and Hibernate*, covers the concept of Object-Relational Mapping, which is introduced using the Hibernate framework. This chapter explains how to execute CRUD operations in Hibernate and fetch strategies and associative mappings and also covers techniques such as caching and pooling for performance optimization.

What you need for this book

In general, PostgreSQL server and client tools do not need an exceptional hardware. PostgreSQL can be installed on almost all modern platforms, including Linux, Windows, and Mac. Also, in the book, when a certain library is needed, the installation instructions are given.

The example provided in this book requires PostgreSQL version 9.4; however, most of the examples can be executed on earlier versions as well. In order to execute the sample code, scripts, and examples provided in the book, you need to have at least a PostgreSQL client tool installed on your machine—preferably psql—and access to a remote server running the PostgreSQL server. In a Windows environment, the cmd.exe command prompt is not very convenient; thus, the user might consider using Cygwin `http://www.cygwin.com/` or another alternative such as Powershell.

For some chapters, such as *Chapter 13*, *PostgreSQL JDBC* and *Chapter 14*, *PostgreSQL and Hibernate*, one needs to install a development kit (JDK). Also, it is convenient to use the NetBeans or Eclipse integrated development environment (IDE).

Who this book is for

If you are a student, database developer, or an administrator interested in developing and maintaining a PostgreSQL database, this book is for you. No knowledge of database programming or administration is necessary.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `customer_service` associates the customer and the service relations."

A block of code is set as follows:

```
<hibernate-mapping package="carportal" schema="carportal_app">
  <class name="Account" table="account">
    <id name="accountID" column="account_id">
      <generator class="identity"/>
    </id>
```

Any command-line input or output is written as follows:

```
SELECT first_name, last_name, service_id
FROM customer AS c CROSS JOIN customer_service AS cs
WHERE c.customer_id=cs.customer_id AND c.customer_id = 3;
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Another option is to use a Linux emulator such as **Cygwin** and **MobaXterm**."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1 Relational Databases

This chapter will provide a high-level overview of topics related to database development. Understanding the basic relational database concepts enables the developers to not only come up with clean designs, but also to master relational databases. This chapter is not restricted to learning PostgreSQL, but covers all relational databases.

The topics covered in this chapter include the following:

- **Database management systems**: Understanding the different database categories enables the developer to utilize the best in each world.
- **Relational algebra**: Understanding relational algebra enables the developers to master the SQL language, especially, SQL code rewriting.
- **Data modeling**: Using data modeling techniques leads to better communication.

Database management systems

Different database management systems support diverse application scenarios, use cases, and requirements. Database management systems have a long history. First we will quickly take a look at the recent history, and then explore the market-dominant database management system categories.

A brief history

Broadly, the term database can be used to present a collection of things. Moreover, this term brings to mind many other terms including data, information, data structure, and management. A database can be defined as a collection or a repository of data, which has a certain structure, managed by a **database management system** (**DBMS**). Data can be structured as tabular data, semi-structured as XML documents, or unstructured data that does not fit a predefined data model.

In early days, databases were mainly aimed at supporting business applications; this led us to the well-defined relational algebra and relational database systems. With the introduction of object-oriented languages, new paradigms of database management systems appeared such as object-relational databases and object-oriented databases. Also, many businesses as well as scientific applications use arrays, images, and spatial data; thus, new models such as raster, map, and array algebra are supported. Graph databases are used to support graph queries such as the shortest path from one node to another along with supporting traversal queries easily.

With the advent of web applications such as social portals, it is now necessary to support a huge number of requests in a distributed manner. This has led to another new paradigm of databases called NoSQL (Not Only SQL) which has different requirements such as performance over fault tolerance and horizontal scaling capabilities.

In general, the timeline of database evolution was greatly affected by many factors such as:

- **Functional requirements**: The nature of the applications using a DBMS has led to the development of extensions on top of relational databases such as PostGIS (for spatial data) or even dedicated DBMS such as SCI-DB (for scientific data analytics).
- **Nonfunctional requirements**: The success of object-oriented programming languages has created new trends such as object-oriented databases. Object relational database management systems have appeared to bridge the gap between relational databases and the object-oriented programming languages. Data explosion and the necessity to handle terabytes of data on commodity hardware have led to columnar databases, which can easily scale up horizontally.

Database categories

Many database models have appeared and vanished such as the network model and hierarchal model. The predominant categories now in the market are relational, object-relational databases, and NoSQL databases. One should not think of NoSQL and SQL databases as rivals; they are complementary to each other. By utilizing different database systems, one can overcome many limitations, and get the best of different technologies.

The NoSQL databases

The NoSQL databases are affected by the CAP theorem, also known as Brewer's theorem. In 2002, S. Gilbert and N. Lynch published a formal proof of the CAP theorem in their article: "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In 2009, the NoSQL movement began. Currently, there are over 150 NoSQL databases (`nosql-database.org`).

The CAP theorem

The CAP theorem states that it is impossible for a distributed computing system to simultaneously provide all three of the following guarantees:

- **Consistency**: All clients see (immediately) the latest data even in the case of updates.
- **Availability**: All clients can find a replica of some data even in the case of a node failure. That means even if some part of the system goes down, the clients can still access the data.
- **Partition tolerance**: The system continues to work regardless of arbitrary message loss or failure of part of the system.

The choice of which feature to discard determines the nature of the system. For example, one could sacrifice consistency to get a scalable, simple, and high-performance database management system.

Often, the main difference between a relational database and a NoSQL database is consistency. A relational database enforces ACID. ACID is the acronym for the following properties: Atomicity, Consistency, Isolation, and Durability. In contrast, many NoSQL databases adopt the **basically available soft-state, eventual-consistency** (**BASE**) model.

NoSQL motivation

A NoSQL database provides a means for data storage, manipulation, and retrieval for non-relational data. The NoSQL databases are distributed, open source and horizontally scalable. NoSQL often adopts the BASE model, which prizes availability over consistency, and informally guarantees that if no new updates are made on a data item, eventually all access to that data item will return the latest version of that data item. The advantages of this approach include the following:

- Simplicity of design
- Horizontal scaling and easy replication
- Schema free
- Huge amount of data support

We will now explore a few types of NoSQL databases.

Key value databases

The key value store is the simplest database store. In this database model, the storage, as its name suggests, is based on maps or hash tables. Some key-value databases allow complex values to be stored as lists and hash tables. Key-value pairs are extremely fast for certain scenarios, but lack the support for complex queries and aggregation. Some of the existing open source key-value databases are Riak, Redis, Memebase, and MemcacheDB.

Columnar databases

Columnar or column-oriented databases are based on columns. Data in a certain column in a two dimensional relation is stored together. Unlike relational databases, adding columns is inexpensive, and is done on a row-by-row basis. Rows can have a different set of columns. Tables can benefit from this structure by eliminating the storage cost of the null values. This model is best suited for distributed databases. HBase is one of the most famous columnar databases. It is based on the Google big table storage system. Column-oriented databases are designed for huge data scenarios, so they scale up easily. For small datasets, HBase is not a suitable architecture. First, the recommended hardware topology for HBase is a five-node or server deployment. Also, it needs a lot of administration, and is difficult to master and learn.

Document databases

A document-oriented database is suitable for documents and semi-structured data. The central concept of a document-oriented database is the notion of a document. Documents encapsulate and encode data (or information) in some standard formats or encodings such as XML, JSON, and BSON. Documents do not adhere to a standard schema or have the same structure; so, they provide a high degree of flexibility. Unlike relational databases, changing the structure of the document is simple, and does not lock the clients from accessing the data.

Document databases merge the power of relational databases and column-oriented databases. They provide support for ad-hoc queries, and can be scaled up easily. Depending on the design of the document database, MongoDB is designed to handle a huge amount of data efficiently. On the other hand, CouchDB provides high availability even in the case of hardware failure.

Graph databases

Graph databases are based on the graph theory, where a database consists of nodes and edges. The nodes as well as the edges can be assigned data. Graph databases allow traversing between the nodes using edges. Since a graph is a generic data structure, graph databases are capable of representing different data. A famous implementation of an open source commercially supported graph databases is Neo4j.

Relational and object relational databases

Relational database management systems are one of the most-used DBMSs in the world. It is highly unlikely that any organization, institution, or personal computer today does not have or use a piece of software that does not rely on RBDMS. Software applications can use relational databases via dedicated database servers or via lightweight RDBMS engines, embedded in the software applications as shared libraries.

The capabilities of a relational database management system vary from one vendor to another, but most of them adhere to the ANSI SQL standards. A relational database is formally described by relational algebra, and is modeled on the relational model. **Object-relational database** (**ORD**) are similar to relational databases. They support object-oriented model concepts such as:

- User defined and complex data types
- Inheritance

ACID properties

In a relational database, a single logical operation is called a transaction. The technical translation of a transaction is a set of database operations, which are create, read, update, and delete (CRUD). The simplest example for explaining a transaction is money transfer from one bank account to another, which normally involves debiting one account and crediting another. The ACID properties in this context could be described as follows:

- **Atomicity**: All or nothing, which means that if a part of a transaction fails, then the transaction fails as a whole.
- **Consistency**: Any transaction gets the database from one valid state to another valid state. Database consistency is governed normally by data constraints and the relation between data and any combination thereof. For example, imagine if one would like to completely purge his account on a shopping service. In order to purge his account, his account details, such as list of addresses, will also need to be purged. This is governed by foreign key constraints, which will be explained in detail in the next chapter.
- **Isolation**: Concurrent execution of transactions results in a system state that would be obtained if the transactions were executed serially.
- **Durability**: The transactions which are committed, that is executed successfully, are persistent even with power loss or some server crashes. This is done normally by a technique called **write-ahead log** (**WAL**).

The SQL Language

Relational databases are often linked to the **Structured Query Language** (**SQL**). SQL is a declarative programming language, and is the standard relational database language. **American National Standard Institute** (**ANSI**) and **International standard organization** (**ISO**) published the SQL standard for the first time in 1986, followed by many versions such as SQL:1999, SQL:2003, SQL:2006, SQL:2008, and so on.

The SQL language has several parts:

- **Data definition language (DDL**): It defines and amends the relational structure.
- **Data manipulation language (DML)**: It retrieves and extracts information from the relations.
- **Data control language (DCL)**: It controls the access rights to relations.

Basic concepts

A relational model is a first-order predicate logic, which was first introduced by Edgar F. Codd. A database is represented as a collection of relations. The state of the whole database is defined by the state of all the relations in the database. Different information can be extracted from the relations by joining and aggregating data from different relations, and by applying filters on the data.

In this section, the basic concepts of the relational model are introduced using the top-down approach by first describing the relation, tuple, attribute, and domain.

The terms relation, tuple, attribute, and unknown, which are used in the formal relational model, are equivalent to table, row, column, and null in the SQL language.

Relation

Think of a relation as a table with a header, columns, and rows. The table name and the header help in interpreting the data in the rows. Each row represents a group of related data, which points to a certain object.

A relation is represented by a set of tuples. Tuples should have the same set of ordered attributes. Attributes have a domain, that is, a type and a name.

Customer relation					
	`customer_id`	`first_name`	`last_name`	`Email`	`Phone`
Tuple →	`1`	`Thomas`	`Baumann`		`6622347`
Tuple →	`2`	`Wang`	`Kim`	`kim@wang_kim.com`	`6622345`
Tuple →	`3`	`Christian`	`Bayer`		`6622919`
Tuple →	`4`	`Ali`	`Ahmad`		`3322123`
	↑ **Attribute**	↑ **Attribute**	↑ **Attribute**	↑ **Attribute**	↑ **Attribute**

The relation schema is denoted by the relation name and the relation attributes. For example, customer (`customer_id`, `first_name`, `last_name`, and `Email`) is the relation schema for the customer relation. Relation state is defined by the set of relation tuples; thus, adding, deleting, and amending a tuple will change the relation to another state.

Tuple order or position in the relation is not important, and the relation is not sensitive to tuple order. The tuples in the relation could be ordered by a single attribute or a set of attributes. Also, a relation cannot have duplicate tuples.

A relation can represent entities in the real world, such as a customer, or can be used to represent an association between relations. For example, the customer could have several services, and a service can be offered to several customers. This could be modeled by three relations: `customer`, `service`, and `customer_service`. The `customer_service` relation associates the customer and the service relations. Separating the data in different relations is a key concept in relational database modeling. This concept called normalization is the process of organizing relation columns and relations to reduce data redundancy. For example, let us assume a collection of services is stored in the customer relation. If a service is assigned to multiple customers, that would result in data redundancy. Also, updating a certain service would require updating all its copies in the customer table.

Tuple

A tuple is a set of ordered attributes. They are written by listing the elements within parentheses `()` and separated by commas, such as (john, smith, 1971). Tuple elements are identified via the attribute name. Tuples have the following properties:

- `(a1,a2, a3, …an) = (b1, b2,b3,…,bn ) if and only if a1 = ba , a2=b2, … an= bn`
- A tuple is not a set, the order of attributes matters.
    - `(a1, a2) ≠(a2, a1)`
    - `(a1, a1) ≠(a1)`
    - A tuple has a finite set of attributes

In the formal relational model, multi-valued attributes as well as composite attributes are not allowed. This is important to reduce data redundancy and increasing data consistency. This isn't strictly true in modern relational database systems because of the utilization of complex data types such as JSON and key-value stores. There is a lot of debate regarding the application of normalization; the rule of thumb is to apply normalization unless there is a good reason not to do so.

Another important concept is that of the unknown values, that is, `NULL` values. For example, in the customer relation, the phone number of a customer might be unknown. Predicates in relational databases uses three-valued logic (3VL), where there are three truth values: true, false, and unknown. In a relational database, the third value, unknown, can be interpreted in many ways, such as unknown data, missing data, or not applicable. The three-valued logic is used to remove ambiguity. Imagine two tuples in the customer relation with missing phone values; does that mean both have the same phone, that is, `NULL=NULL`? The evaluation of the expression `NULL=NULL` is also `NULL`.

		A		
		True	**False**	**Unknown**
B	**True**	True	True	True
	False	True	False	Unknown
	Unknown	True	Unknown	Unknown

Logical operator OR truth table

		A		
		True	**False**	**Unknown**
B	**True**	True	False	Unknown
	False	False	False	Unknown
	Unknown	True	False	Unknown

Logical AND truth table

A	**True**	**False**	**Unknown**
NOT A	False	True	Unknown

Logical NOT truth table

Attribute

Each attribute has a name and a domain, and the name should be distinct within the relation. The domain defines the possible set of values that the attribute can have. One way to define the domain is to define the data type and a constraint on this data type. For example, hourly wage should be a positive real number and bigger than five if we assume the minimum hourly wage is five dollars. The domain could be continuous, such as salary which is any positive real number, or discrete, such as gender.

The formal relational model puts a constraint on the domain: the value should be atomic. Atomic means that each value in the domain is indivisible. For instance, the `name` attribute domain is not atomic, because it can be divided into first name and last name. Some examples of domains are as follows:

- **Phone number**: Numeric text with a certain length.
- **Country code**: Defined by ISO 3166 as a list of two letter codes (ISO alpha-2) and three letter codes (ISO alpha-3). The country codes for Germany are DE and DEU for alpha-2 and alpha-3 respectively.

[It is good practice if you have lookup tables such as country code, currency code, and languages to use the already defined codes in ISO standards, instead of inventing your own codes.]

Constraint

The relational model defines many constraints in order to control data integrity, redundancy, and validity.

- **Redundancy**: Duplicate tuples are not allowed in the relation.
- **Validity**: Domain constraints control data validity.
- **Integrity**: The relations within a single database are linked to each other. An action on a relation such as updating or deleting a tuple might leave the other relations in an invalid state.

We could classify the constraints in a relational database roughly into two categories:

- **Inherited constraints from the relational model**: Domain integrity, entity integrity, and referential integrity constraints.
- **Semantic constraint, business rules, and application specific constraints**: These constraints cannot be expressed explicitly by the relational model. However, with the introduction of procedural SQL languages such as PL/pgsql for PostgreSQL, relational databases can also be used to model these constraints.

Domain integrity constraint

The domain integrity constraint ensures data validity. The first step in defining the domain integrity constraint is to determine the appropriate data type. The domain data types could be `integer`, `real`, `boolean`, `character`, `text`, `inet`, and so on. For example, the data type of first name and e-mail address is text. After specifying the data type, check constraints, such as the mail address pattern, need to be defined.

- **Check constraint**: A check constraint can be applied to a single attribute or a combination of many attributes in a tuple. Let us assume that `customer_service` schema is defined as (`customr_id`, `service_id`, `start_date`, `end _date`, `order_date`). For this relation, we can have a check constraint to make sure that `start_date` and `end_date` are entered correctly by applying the following check (`start_date<end_date`).
- **Default constraint**: The attribute can have a default value. The default value could be a fixed value such as the default hourly wage of the employees , for example, $10. It may also have a dynamic value based on a function such as random, current time, and date. For example, in the `customer_service` relation, `order_date` can have a default value which is the current date.
- **Unique constraint**: A unique constraint guarantees that the attribute has a distinct value in each tuple. It allows null values. For example, let us assume we have a relation player defined as player (`player_id`, `player_nickname`). The player uses his ID to play with others; he can also pick up a nickname which is not used by someone else.
- **Not null constraint**: By default, the attribute value can be null. The not null constraint restricts an attribute from having a null value. For example, each person in the birth registry record should have a name.

Entity integrity constraint

In the relational model, a relation is defined as a set of tuples. By definition, the element of the set is distinct. This means that all the tuples in a relation must be distinct. The entity integrity constraint is enforced by having a primary key which is an attribute/set of attributes having the following characteristics:

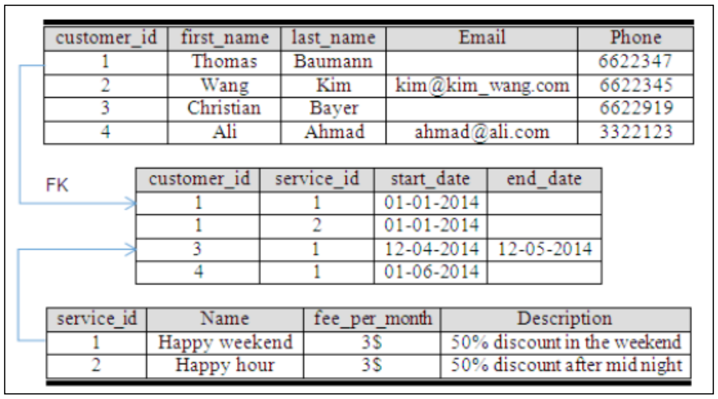
- The attribute should be unique
- The attributes should be not null

Each relation must have only one primary key, but can have many unique keys. A candidate key is a minimal set of attributes which can identify a tuple. All unique, `not null` attributes can be candidate keys. The set of all attributes form a super key. In practice, we often pick up a single attribute to be a primary key instead of a compound key (key that consists of two or more attributes that uniquely identify a tuple) to reduce data redundancy, and to ease the joining of the relations with each other.

If the primary key is generated by the DBMS, then it is called a surrogate key. Otherwise, it is called a natural key. The surrogate key candidates can be sequences and **universal unique identifiers** (**UUID**). A surrogate key has many advantages such as performance, requirement change tolerance, agility, and compatibility with object relational mappers. More on surrogate keys will be covered in the following chapters.

Referential integrity constraints

Relations are associated with each other via common attributes. Referential integrity constraints govern the association between two relations, and ensure data consistency between tuples. If a tuple in one relation references a tuple in another relation, then the referenced tuple must exist. In the customer service example, if a service is assigned to a customer, then the service and the customer must exist as shown in the following example. For instance, in the `customer_service` relation, we cannot have a tuple with values (`5`, `1`,`01-01-2014`, `NULL`), because we do not have a customer with `customer_id` equal to 5.



customer_id	first_name	last_name	Email	Phone
1	Thomas	Baumann		6622347
2	Wang	Kim	kim@kim_wang.com	6622345
3	Christian	Bayer		6622919
4	Ali	Ahmad	ahmad@ali.com	3322123

customer_id	service_id	start_date	end_date
1	1	01-01-2014	
1	2	01-01-2014	
3	1	12-04-2014	12-05-2014
4	1	01-06-2014	

service_id	Name	fee_per_month	Description
1	Happy weekend	3$	50% discount in the weekend
2	Happy hour	3$	50% discount after mid night

Association between customer and service

The lack of referential integrity constraints can lead to many problems such as:

- Invalid data in the common attributes
- Invalid information during joining of data from different relations.

Referential integrity constraints are achieved via foreign keys. A foreign key is an attribute or a set of attributes that can identify a tuple in the referenced relation. Since the purpose of a foreign key is to identify a tuple in the referenced relation, foreign keys are generally primary keys in the referenced relation. Unlike a primary key, a foreign key can have a null value. It can also reference a unique attribute in the referenced relation. Allowing a foreign key to have a null value enables us to model different cardinality constraints. Cardinality constraints define the participation between two different relations. For example, a parent can have more than one child; this relation is called one-to-many relationship, because one tuple in the referenced relation is associated with many tuples in the referencing relation. Also, a relation could reference itself. This foreign key is called a self-referencing or recursive foreign key. For example, a company acquired by another company:

`company_id`	`Name`	`acquisitioned_by`
`1`	`Facebook`	
`2`	`WhatsApp`	`1`
Primary key		**Foreign key**

Recursive foreign key

To ensure data integrity, foreign keys can be used to define several behaviors when a tuple in the referenced relation is updated or deleted. The following behaviors are called referential actions:

- **Cascade**: When a tuple is deleted or updated in the referenced relation, the tuples in the referencing relation are also updated or deleted.
- **Restrict**: The tuple cannot be deleted or the referenced attribute cannot be updated if it is referenced by another relation.
- **No action**: Similar to restrict, but it is deferred to the end of the transaction.
- **Set default**: When a tuple in the referenced relation is deleted or the referenced attribute is updated, then the foreign key value is assigned the default value.
- **Set null**: The foreign key attribute value is set to null when the referenced tuple is deleted.

Semantic constraints

Semantic integrity constraints or business logic constraints describe the database application constraints in general. Those constraints are either enforced by the business logic tier of the application program or by SQL procedural languages. Trigger and rule systems can also be used for this purpose. For example, the customer should have at most one active service at a time. Based on the nature of the application, one could favor using an SQL procedural language or a high-level programming language to meet the semantic constraints. The advantages of using the SQL programming language are:

- **Performance**: RDBMSs often have complex analyzers to generate efficient execution plans. Also, in some cases such as data mining, the amount of data that needs to be manipulated is very large. Manipulating the data using procedural SQL language eliminates the network data transfer. Finally, some procedural SQL languages utilize clever caching algorithms.
- **Last minute change**: For the SQL procedural languages, one could deploy bug fixes without service disruption.

Relational algebra

Relational algebra is the formal language of the relational model. It defines a set of closed operations over relations, that is, the result of each operation is a new relation. Relational algebra inherits many operators from set algebra. Relational algebra operations could be categorized into two groups:

- The first one is a group of operations which are inherited from set theory such as `UNION`, `INTERSECTION`, `SET DIFFERENCE`, and `CARTESIAN PRODUCT`, also known as `CROSS PRODUCT`.
- The second is a group of operations which are specific to the relational model such as `SELECT` and `PROJECT`.

Relational algebra operations could also be classified as binary and unary operations. Primitive relational algebra operators have ultimate power of reconstructing complex queries. The primitive operators are:

- `SELECT` (σ): A unary operation written as $\sigma_{\varphi}\mathrm{RO}$ where φ is a predicate. The selection retrieves the tuples in R, where φ holds.

- `PROJECT` (π): A unary operation used to slice the relation in a vertical dimension, that is, attributes. This operation is written as $\pi_{\mathrm{a1,a2\ldots an}}\mathrm{RO}$, where $a1, a2, \ldots, an$ are a set of attribute names.
- `CARTESIAN PRODUCT` ($\times$): A binary operation used to generate a more complex relation by joining each tuple of its operands together. Let us assume that *R* and *S* are two relations, then $R \times S = \{r1, r2, \ldots, rn, s1, s2, \ldots, sn\}$, where $r1, r2, \ldots, rn \in R$ and $s1, s2, \ldots, sn \in S$.
- `UNION` ($\cup$): Appends two relations together; note that the relations should be union compatible, that is, they should have the same set of ordered attributes. Formally, $R \cup S = (r1, r2, \ldots, rn) \cup (s1, s2, \ldots, sn)$, where $(r1, r2, \ldots, rn) \in R$ and $(s1, s2, \ldots, sn) \in R$.
- `DIFFERENCE` ($/$ or `-`): A binary operation in which the operands should be union compatible. Difference creates a new relation from the tuples, which exist in one relation but not in the other. The set difference for the relation R and S can be given as $R / S = (r1, r2, \ldots, rn)$, where $(r1, r2, \ldots, rn) \in R$ and $(r1, r2, \ldots, rn) \notin S$.
- `RENAME` (ρ): A unary operation that works on attributes. It simply renames an attribute. This operator is mainly used in JOIN operations to distinguish the attributes with the same names but in different relation tuples. Rename is expressed as $\rho_{a/b}R$.

In addition to the primitive operators, there are aggregation functions such as `sum`, `count`, `min`, `max`, and `average` aggregates. Primitive operators can be used to define other relation operators such as left-join, right-join, equi-join, and intersection. Relational algebra is very important due to its expressive power in optimizing and rewriting queries. For example, the selection is commutative, so $\sigma_a \sigma_b R = \sigma_b \sigma_a R$. A cascaded selection may also be replaced by a single selection with a conjunction of all the predicates, that is, $\sigma_a \sigma_b R = \sigma_{a\,AND b} R$.

The SELECT and PROJECT operations

`SELECT` is used to restrict tuples from the relation. If no predicate is given then the whole set of tuples is returned. For example, the query "give me the customer information where the `customer_id` equals to `2`" is written as:

$$\sigma_{customer_{id}=2}\ customer$$

The selection is commutative; the query "give me all customers where the customer mail is known, and the customer first name is `kim`" is written in three different ways, as follows:

$$\sigma_{emails\ is\ not\ null}\left(\sigma_{first\_name=kim} customer\right)$$

$$\sigma_{first\_name=kim}\left(\sigma_{emails\ is\ not\ null}\ customer\right)$$

$$\sigma_{first\_name=kim\ AND\ emails\ is\ not\ null}\ customer$$

The selection predicates are certainly determined by the data types. For numeric data types, the comparison operator might be ($\neq, =<, >, \geq, \leq$). The predicate expression can contain complex expressions and functions.

The equivalent SQL statement for the `SELECT` operator is the `SELECT *` statement, and the predicate is defined in the `WHERE` clause. Finally, the `*` means all the relation attributes; note that in the production environment, it is not recommended to use `*`. Instead, one should list all the relation attributes explicitly.

```
SELECT * FROM customer WHERE customer_id=2
```

The project operation could be visualized as vertical slicing of the table. The query: "give me the customer names" is written in relational algebra as follows:

$$\pi_{first\_name,last\_name} Customer$$

first_name	**last_name**
Thomas	Baumann
Wang	Kim
Christian	Bayer
Ali	Ahmad

The result of project operation

Duplicate tuples are not allowed in the formal relational model; the number of returned tuples from the project operator is always equal to or less than the number of total tuples in the relation. If a project operator's attribute list contains a primary key, then the resulting relation has the same number of tuples as the projected relation.

Cascading projections could be optimized as the following expression:

$$\pi_{att1}\left(\pi_{att1,}\pi_{att2}R\right)=\pi_{att1}R$$

The SQL equivalent for the `PROJECT` operator in SQL is `SELECT DISTINCT`. The `DISTINCT` keyword is used to eliminate duplicates. To get the result shown in the preceding expression, one could execute the following SQL statement:

```
SELECT DISTINCT first_name, last_name FROM customers;
```

The sequence of the execution of the `PROJECT` and `SELECT` operations can be interchangeable in some cases.

The query "give me the name of the customer with `customer_id` equal to `2`" could be written as:

$$\sigma_{customer\_id=2}\left(\pi_{first\_name,last\_name}\,customer\right)$$

$$\pi_{first\_name,last\_name}\left(\sigma_{customer\_id=2}\,customer\right)$$

In other cases, the `PROJECT` and `SELECT` operators must have an explicit order as shown in the following example; otherwise, it will lead to an incorrect expression. The query "give me the last name of the customers where the first name is `kim`" could be written as the following expression:

$$\pi_{last\_name}\left(\sigma_{first\_name=kim}\,customer\right)$$

The RENAME operation

The `Rename` operation is used to alter the attribute name of the resultant relation, or to give a specific name to the resultant relation. The `Rename` operation is used to:

- Remove confusion if two or more relations have attributes with the same name
- Provide user-friendly names for attributes, especially when interfacing with reporting engines
- Provide a convenient way to change the relation definition, and still be backward compatible

The `AS` keyword in SQL is the equivalent of the `RENAME` operator in relational algebra. the following SQL example creates a relation with one tuple and one attribute, which is renamed `PI`.

```
SELECT 3.14::real AS PI;
```

The Set theory operations

The set theory operations are union, intersection, and minus (difference). Intersection is not a primitive relational algebra operator, because it is can be written using the union and difference operators:

$$A \cap B = \left(\left(A \cup B\right) - \left(A - B\right)\right) - \left(B - A\right)$$

The intersection and union are commutative:

$$A \cup B = \left(B \cup A\right)$$

$$A \cap B = \left(B \cap A\right)$$

For example, the query "give me all the customer IDs where the customer does not have a service assigned to him" could be written as:

$$\sigma_{customer\_id} customer - \sigma_{customer\_id} customer\_service$$

The CROSS JOIN (Cartesian product) operation

The CROSS JOIN operation is used to combine tuples from two relations into a single relation. The number of attributes in a single relation equals the sum of the number of attributes of the two relations. The number of tuples in the single relation equals the product of the number of tuples in the two relations. Let us assume *A* and *B* are two relations, and $C = A \times B$. Then:

$$number\ of\ attribute(C) = number\ of\ attributes(A) + number\ of\ attributes(B)$$

$$number\ of\ tuples(C) = number\ of\ tuples(A) * number\ of\ tuples(B)$$

The following image shows the cross join of customer and customer service, that is, $customer \times customer\_service$:

customer.customer_id	first_name	last_name	Email	phone	customer_service.customer_id	service_id	start_date	end_date
1	Thomas	Baumann		6622347	1	1	01-01-2014	
2	Wang	Kim	kim@kim_wang.com	6622345	1	1	01-01-2014	
3	Christian	Bayer		6622919	1	1	01-01-2014	
4	Ali	Ahmad	ahmad@ali.com	3322123	1	1	01-01-2014	
1	Thomas	Baumann		6622347	1	2	01-01-2014	
2	Wang	Kim	kim@kim_wang.com	6622345	1	2	01-01-2014	
3	Christian	Bayer		6622919	1	2	01-01-2014	

customer.customer_id	first_name	last_name	Email	phone	customer_service.customer_id	service_id	start_date	end_date
4	Ali	Ahmad	ahmad@ali.com	3322123	1	2	01-01-2014	
1	Thomas	Baumann		6622347	3	1	12-04-2014	12-05-2014
2	Wang	Kim	kim@kim_wang.com	6622345	3	1	12-04-2014	12-05-2014
3	Christian	Bayer		6622919	3	1	12-04-2014	12-05-2014
4	Ali	Ahmad	ahmad@ali.com	3322123	3	1	12-04-2014	12-05-2014
1	Thomas	Baumann		6622347	4	1	01-06-2014	
2	Wang	Kim	kim@kim_wang.com	6622345	4	1	01-06-2014	
3	Christian	Bayer		6622919	4	1	01-06-2014	
4	Ali	Ahmad	ahmad@ali.com	3322123	4	1	01-06-2014	

CROSS JOIN of customer and customer_service relations

For example, the query "for the customer with `customer_id` equal to `3`, retrieve the customer name and the customer service IDs" could be written in SQL as follows:

```
SELECT first_name, last_name, service_id
FROM customer AS c CROSS JOIN customer_service AS cs
WHERE c.customer_id=cs.customer_id AND c.customer_id = 3;
```

In the preceding example, one can see the relationship between relational algebra and the SQL language. It shows how relational algebra could be used to optimize query execution. This example could be executed in several ways, such as: