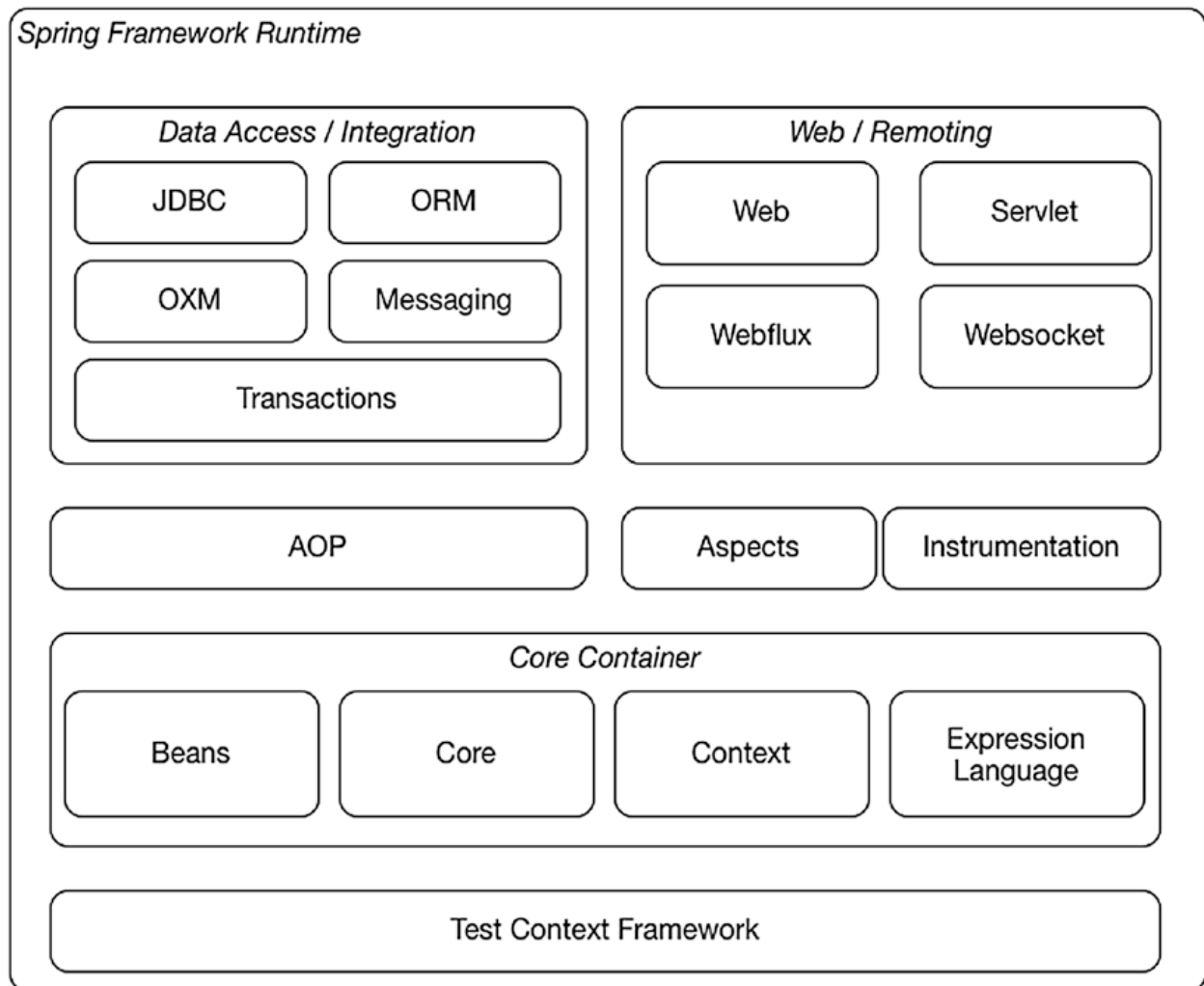


Spring MVC + WebFlux + Oracle



Module Artifact Description

AOP spring-aop The proxy-based AOP framework for Spring

Aspects spring-aspects AspectJ-based aspects for Spring

Beans spring-beans Spring's core bean factory support

Context spring-context Application context runtime implementations; also contains scheduling and remoting support classes

Context spring-context-indexer

Support for providing a static index of beans used in the application; improves startup performance

Context spring-context-support

Support classes for integrating third-party libraries with Spring

Core spring-core Core utilities

Expression

Language

spring-expression

Classes for the Spring Expression Language (SpEL)

Instrumentation spring-instrument

Instrumentation classes to be used with a Java agent

JCL spring-jcl Spring specific replacement for commons-logging

JDBC spring-jdbc JDBC support package that includes datasource setup classes and JDBC access support

JMS spring-jms JMS support package that includes synchronous JMS access and message listener containers

ORM spring-orm ORM support package that includes support for Hibernate 5+ and JPA

Messaging spring-messaging

Spring messaging abstraction; used by JMS and WebSocket

OXM spring-oxm XML support package that includes support for object-to-XML mapping; also includes support for JAXB, JiBX, XStream, and Castor

Test spring-test Testing support classes

Transactions spring-tx Transaction infrastructure classes; includes JCA integration and DAO support classes

Web spring-web Core web package for use in any web environment

WebFlux spring-webflux Spring WebFlux support package Includes support for several reactive runtimes like Netty and Undertow

Servlet spring-webmvc Spring MVC support package for use in a Servlet environment Includes support for common view technologies

WebSocket spring-websocket

Spring WebSocket support package Includes support for communication over the WebSocket protocol

Dependency Injection

In dependency injection (DI), objects are given their dependencies at construction time.

It is a Spring Framework foundation. You have probably heard of Inversion of Control (IoC).⁷ IoC is a broader, more general concept that can be addressed in different ways.

IoC lets developers decouple and focus on what is important for a given part of an enterprise application, but without thinking about what other parts of the system do.

Programming to interfaces is one way to think about decoupling.

Almost every enterprise application consists of multiple components that need to work together. In the early days of Java enterprise development, we simply put all the logic of constructing those objects (and the objects they needed) in the constructor (see Listing 2-1). At first sight, there is nothing wrong with that approach; however, as time progresses, object construction became slow, and objects had a lot of knowledge they shouldn't have had (see the single-responsibility principle).⁸ Those classes became hard to maintain, and they were also hard to the unit and/or integration test.

ApplicationContexts

To do dependency injection in Spring, you need an application context. In Spring, this is an instance of the `org.springframework.context.ApplicationContext` interface. The

application context is responsible for managing the beans defined in it. It also enables more elaborate things like applying AOP to the beans defined in it.

An ApplicationContext | Overview Implementation | Location File type
ClassPathXmlApplicationContext Classpath XML
FileSystemXmlApplicationContext File system XML
AnnotationConfigApplicationContext Classpath Java
XmlWebApplicationContext Web Application Root XML
AnnotationConfigWebApplicationContext Web Application Classpath Java

Resource Loading

Prefix Location

classpath: The root of the classpath

file: File system

http: Web application root

Expression Description

classpath:/META-INF/

spring/*.xml

Loads all files with the XML file extensions from the classpath in the META-INF/spring directory

file:/var/conf/*.

properties

Loads all files with the properties file extension from the /var/conf directory and all subdirectories

Component Scanning

Spring also has something called component scanning. In short, this feature enables Spring to scan your classpath for classes that are annotated with org.springframework.stereotype.Component (or one of the specialized annotations like @Service, @Repository, @Controller, or org.springframework.context.annotation.Configuration). If we want to enable component scanning, we need to instruct the application context to do so. The org.springframework.context.annotation.ComponentScan annotation enables us to accomplish that. This annotation needs to be put on our configuration class to enable component scanning.

@Configuration

```
@ComponentScan(basePackages = {  
    "com.apress.prospringmvc.moneytransfer.scanning",  
    "com.apress.prospringmvc.moneytransfer.repository" })  
public class ApplicationContextConfiguration {}
```

Scopes

By default, all beans in a Spring application context are singletons. As the name implies,

there is a single instance of a bean, and it is used for the whole application. This doesn't typically present a problem because our services and repositories don't hold state; they simply execute a certain operation and (optionally) return a value.

An Overview of Scopes

Prefix Description

singleton The default scope. A single instance of a bean is created and shared throughout the application.

prototype Each time a certain bean is needed, a fresh instance of the bean is returned.

thread The bean is created when needed and bound to the currently executing thread.

If the thread dies, the bean is destroyed.

request The bean is created when needed and bound to the lifetime of the incoming `javax.servlet.ServletRequest`. If the request is over, the bean instance is destroyed.

session The bean is created when needed and stored in `javax.servlet`.

`HttpSession`. When the session is destroyed, so is the bean instance.

globalSession The bean is created when needed and stored in the globally available session (which is available in Portlet environments). If no such session is available, the scope reverts to the session scope functionality.

application This scope is very similar to the singleton scope; however, beans with this scope are also registered in `javax.servlet.ServletContext`.

Profiles

Spring introduced profiles in version 3.1. Profiles make it easy to create different configurations of our application for different environments. For instance, we can create separate profiles for our local environment, testing, and our deployment to CloudFoundry. Each of these environments requires some environment-specific configuration or beans. You can think of database configuration, messaging solutions, and testing environments, stubs of certain beans. To enable a profile, we need to tell the application context in which profiles are active.

@Configuration

```
public class ApplicationContextConfiguration {  
    @Bean  
    public AccountRepository accountRepository() {  
        return new MapBasedAccountRepository();  
    }  
    @Bean  
    public MoneyTransferService moneyTransferService() {  
        return new MoneyTransferServiceImpl();  
    }  
}  
@Configuration  
@Profile(value = "test")  
public static class TestContextConfiguration {  
    @Bean
```

```

public TransactionRepository transactionRepository() {
return new StubTransactionRepository();
}
}
@Configuration
@Profile(value = "local")
public static class LocalContextConfiguration {
@Bean
public TransactionRepository transactionRepository() {
return new MapBasedTransactionRepository();
}
}
}
}

```

Enabling Features

The Spring Framework offers a lot more flexibility than dependency injection; it also provides many different features we can enable. We can enable these features using annotations (see Table 2-6). Note that we won't use all the annotations in Table 2-6; however, our sample application uses transactions, and we use some AOP. The largest part of this book is about the features provided by the `org.springframework.web.servlet.config.annotation.EnableWebMvc` and `org.springframework.web.reactive.config.EnableWebFlux` annotations.

Spring Boot automatically enables some of these features; it depends on the classes detected on the classpath.

Annotation Description Detected

by Spring

Boot

`org.springframework.
context.annotation.`

`EnableAspectJAutoProxy`

Enables support for handling beans

stereotyped as `org.aspectj.lang.annotation.`

`Aspect.`

Yes

`org.springframework.
scheduling.annotation.`

`EnableAsync`

Enables support for handling bean methods
with the `org.springframework.`

`scheduling.annotation.Async` or

`javax.ejb.Asynchronous` annotations.

No

`org.springframework.cache.`

annotation.EnableCaching

Enables support for bean methods with the
org.springframework.cache.annotation.

Cacheable annotation.

Yes

org.springframework.

context.annotation.

EnableLoadTimeWeaving

Enables support for load-time weaving. By
default, Spring uses a proxy-based approach
to AOP; however, this annotation enables us
to switch to load-time weaving. Some JPA
providers require it.

No

org.springframework.

scheduling.annotation.

EnableScheduling

Enables support for annotation-driven
scheduling, letting us parse bean methods
annotated with the org.springframework.
scheduling.annotation.Scheduled annotation.

No

org.springframework.

beans.factory.aspectj.

EnableSpringConfigured

Enables support for applying dependency
injection to non-Spring managed beans. In
general, such beans are annotated with the
org.springframework.beans.factory.
annotation.Configurable annotation.

This feature requires load-time or compiletime
weaving because it needs to modify
class files.

No

org.springframework.

transaction.annotation.

EnableTransactionManagement

Enables annotation-driven transaction
support, using org.springframework.
transaction.annotation.

Transactional or javax.ejb.

TransactionAttribute to drive
transactions.

Yes

org.springframework.web.
servlet.config.annotation.

EnableWebMvc

Enables support for the powerful and flexible
annotation-driven controllers with request
handling methods. This feature detects
beans with the org.springframework.
stereotype.Controller annotation
and binds methods with the org.
springframework.web.bind.
annotation.RequestMapping annotations
to URLs.

Yes

org.springframework.web.
reactive.config.EnableWebFlux

Enables support for the powerful and flexible
reactive web implementation using the well-known
concepts from Spring Web MVC and,
where possible, extend on them.

Yes

Aspect-Oriented Programming

To enable the features listed in Table 2-4, Spring uses aspect-oriented programming (AOP). AOP is another way of thinking about the structure of software. It enables you to modularize things like transaction management or performance logging, features that span multiple types and objects (cross-cutting concerns). In AOP, there are a few important concepts to keep in mind (see Table 2-7).

Now let's look at transaction management and how Spring uses AOP to apply transactions around methods. The transaction advice, or interceptor, is org.springframework.transaction.interceptor.TransactionInterceptor. This advice is placed around methods with the org.springframework.transaction.annotation.

Transactional annotation. To do this, Spring creates a wrapper around the actual object, which is known as a proxy (see Figure 2-5). A proxy acts like an enclosing object, but it allows (dynamic) behavior to be added (in this case, the transactionality of the method).

Table 2-7. Core AOP Concepts

Concept Description

Aspect The modularization of a cross-cutting concern. In general, this is a Java class with the org.aspectj.lang.annotation.Aspect annotation.

Join Point A point during the execution of a program. This can be the execution of a method, the assignment of a field, or the handling of an exception. In Spring, a join point is always the execution of a method!

Advice The specific action taken by an aspect at a particular join point. There are several types of advice: before, after, after throwing, after returning, and around. In Spring, an advice is called an interceptor because we are intercepting method invocations.

Pointcut A predicate that matches join points. The advice is associated with a pointcut

expression and runs at any join point matching the pointcut. Spring uses the AspectJ expression language by default. Join points can be written using the `org.aspectj.lang.annotation.Pointcut` annotation.

Spring Boot

All the things mentioned previously in this chapter also apply to Spring Boot. Spring Boot builds upon and extends the features of the Spring Framework. It does make things a lot easier, however. Spring Boot automatically configures the features it finds on the classpath by default. When Spring Boot detects Spring MVC classes, it starts Spring MVC. When it finds a `DataSource` implementation, it bootstraps it.

MVC in Short

Component Description

Model The model is the data needed by the view so that it can be rendered. It might be an order placed or a list of books requested by a user.

View The view is the actual implementation, and it uses the model to render itself in a web application. This could be a JSP or JSF page, but it could also be a PDF, XML, or JSON representation of a resource.

Controller The controller is the component that is responsible for responding to the action the user takes, such as form submission or clicking a link. The controller updates the model and takes other actions needed, such as invoking a service method to place an order.

Application Layering

In the introduction, we mentioned that an application consists of several layers (see Figure 3-3). We like to think of a layer as an area of concern for the application. Therefore, we also use layering to achieve separation of concerns. For example, the view shouldn't be burdened with business or data access logic because these are all different concerns and typically located in different layers.

Layer Description

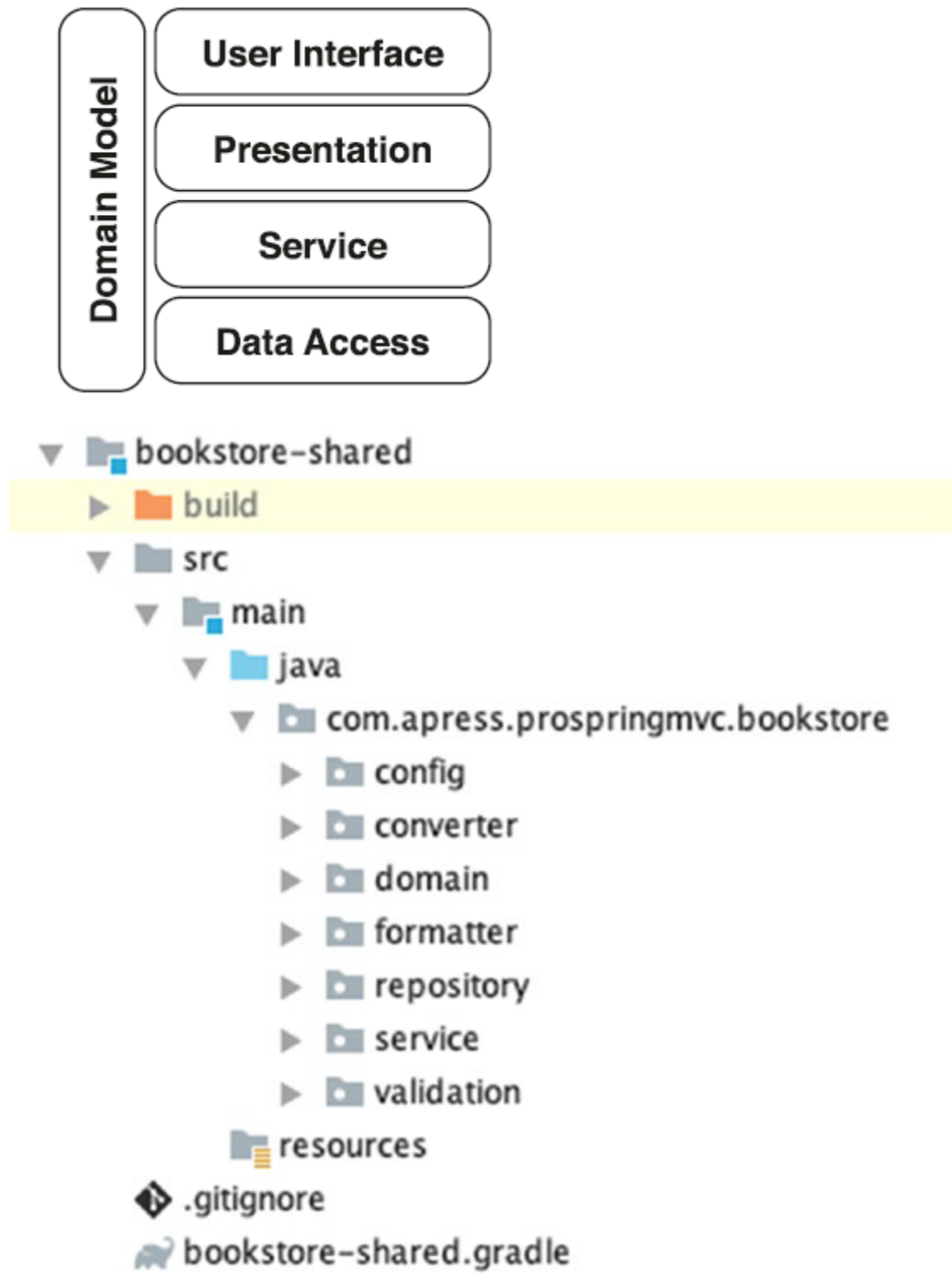
Presentation This is most likely to be a web-based solution. The presentation layer should be as thin as possible. It should also be possible to provide alternative presentation layers like a web frontend or a web service façade. This should all operate on a well-designed service layer.

Service The entry point to the actual system containing the business logic. It provides a coarse-grained interface that enables the use of the system. It is also the layer that should be the system's transactional boundary (and probably security, too). This layer shouldn't know anything (or as little as possible) about persistence or the view technology used.

Data Access An interface-based layer provides access to the underlying data access technology

without exposing it to the upper layers. This layer abstracts the actual persistence framework (e.g., JDBC, JPA, or something like MongoDB). Note that this layer

should not contain business logic.



The Domain Layer

The domain is the most important layer in an application. It is the code representation

of the business problem we are solving, and it contains the business rules of our domain. These rules might check whether we have sufficient funds to transfer money from our account or ensure that fields are unique (e.g., usernames in our system).

The User Interface Layer

The user interface layer presents the application to the user. This layer renders the response generated by the server into the type requested by the user's client. For instance, a web browser will probably request an HTML document, a web service may want an XML document, and another client could request a PDF or Excel document.

The Web Layer

The web layer has two responsibilities. The first responsibility is to guide the user through the web application. The second is to be the integration layer between the service layer and HTTP.

Navigating the user through the website can be as simple as mapping a URL to views or a full-blown page flow solution like Spring Web Flow. The navigation is typically bound to the web layer only, and there isn't any navigation logic in the domain or service layer.

As an integration layer, the web layer should be as thin as possible. It should be the layer that converts the incoming HTTP request to something that can be handled by the service layer and then transforms the result (if any) from the server into a response for the user interface. The web layer should not contain any business logic—that is the sole purpose of the service layer.

The web layer also consists of cookies, HTTP headers, and possibly an HTTP session. It is the responsibility of the web layer to manage all these things consistently and transparently. The different HTTP elements should never creep into our service layer.

The Service Layer

The service layer is very important in the architecture of an application. It is considered the heart of our application because it exposes the system's functionality (the use cases) to the user. It does this by providing a coarse-grained API (as mentioned in Table 3-2).

Listing 3-1 describes a coarse-grained service interface.

Listing 3-1. A Coarse-Grained Service Interface

```
package com.apress.prospringmvc.bookstore.service;
import com.apress.prospringmvc.bookstore.domain.Account;
public interface AccountService {
    Account save(Account account);
    Account login(String username, String password) throws
        AuthenticationException;
    Account getAccount(String username);
}
```

This listing is considered coarse-grained because it takes a simple method call from the client to complete a single use case. This contrasts with the code in Listing 3-2

(fine-grained service methods), which requires a couple of calls to perform a use case.

Listing 3-2. A Fine-Grained Service Interface

```
package com.apress.prospringmvc.bookstore.service;
import com.apress.prospringmvc.bookstore.domain.Account;
public interface AccountService {
    Account save(Account account);
    Account getAccount(String username);
    void checkPassword(Account account, String password);
    void updateLastLogin(Account account);
}
```

The Data Access Layer

The data access layer is responsible for interfacing with the underlying persistence mechanism. This layer knows how to store and retrieve objects from the datastore. It does this in such a way that the service layer doesn't know which underlying datastore is used. (The datastore could be a database, but it could also consist of flat files on the file system.)

There are several reasons for creating a separate data access layer. First, we don't want to burden the service layer with knowledge of the kind of datastore (or datastores) we use; we want to transparently handle persistence. In our sample application, we use an in-memory database and JPA (Java Persistence API) to store our data. Now imagine that, instead of coming from the database, our `com.apress.prospringmvc.bookstore.domain.Account` comes from an Active Directory Service. We could simply create a new implementation of the interface that knows how to deal with Active Directory—all without changing our service layer. In theory, we could easily swap out implementations; for example, we could switch from JDBC to Hibernate without changing the service layer. It is unlikely that this will happen, but it is nice to have this ability.

DispatcherServlet Request Processing Workflow

In the previous chapter, you learned about the important role the front controller plays in a Model 2 MVC pattern. The front controller takes care of dispatching incoming requests to the correct handler and prepares the response to be rendered into something that the user would like to see. The role of the front controller in Spring MVC is played by `org.springframework.web.servlet.DispatcherServlet`. This servlet uses several components to fulfill its role. All these components are expressed as interfaces, for which one or more implementations are available. The next section explores the general role these components play in the request processing workflow. Another upcoming section covers the different implementations of the interfaces.

The DispatcherServlet Components Used in Request Processing Workflow

Component Type Description

```
org.springframework.web.multipart.
MultipartResolver
```

Strategy interface to handle multipart form processing

org.springframework.web.servlet.

LocaleResolver

Strategy for locale resolution and modification

org.springframework.web.servlet.

ThemeResolver

Strategy for theming resolution and modification

org.springframework.web.servlet.

HandlerMapping

Strategy to map incoming requests to handler objects

org.springframework.web.servlet.

HandlerAdapter

Strategy for the handler object type to execute the handler

org.springframework.web.servlet.

HandlerExceptionResolver

Strategy to handle exceptions thrown during handler execution

org.springframework.web.servlet.

RequestToViewNameTranslator

Strategy to determine a view name when the handler returns none

org.springframework.web.servlet.

ViewResolver

Strategy to translate the view name to an actual view implementation

org.springframework.web.servlet.

FlashMapManager

Strategy to simulate flash scope

Bootstrapping DispatcherServlet

The servlet specification (as of version 3.0) has several options for configuring and registering a servlet.

- Option 1: Use a web.xml file (see Listing 4-1).
- Option 2: Use a web-fragment.xml file (see Listing 4-2).
- Option 3: Use javax.servlet.ServletContainerInitializer (see Listing 4-3).
- Option 4: The sample application uses Spring 5.2, so you can get a fourth option by implementing the org.springframework.web.WebApplicationInitializer interface.
- Option 5: Use Spring Boot to autoconfigure DispatcherServlet.

The web.xml Configuration (Servlet 4.0)

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
version="4.0" metadata-complete="true">
<servlet>
<servlet-name>bookstore</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>bookstore</servlet-name>
<url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Using web-fragment.xml

The web-fragment.xml feature has been available since the 3.0 version of the servlet specification, and it allows a more modularized configuration of the web application. The web-fragment.xml has to be in the META-INF directory of a jar file. It isn't detected in the web application's META-INF; it must be in a jar file. web-fragment.xml can contain the same elements as web.xml (see Listing 4-2).

The benefit of this approach is that each module packaged as a jar file contributes to the configuration of the web application. This is also considered a drawback because now you have scattered your configuration over your code base, which could be troublesome in larger projects.

Listing 4-2. The web-fragment.xml Configuration (Servlet 4.0)

```
<web-fragment xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-fragment_4_0.xsd"
version="4.0" metadata-complete="true">
<servlet>
<servlet-name>bookstore</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</
servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>bookstore</servlet-name>
<url-pattern>/*</url-pattern>
```

```
</servlet-mapping>
</web-fragment>
```

Using ServletContainerInitializer

The 3.0 version of the servlet specification introduced the option to use a Java-based approach to configuring your web environment (see Listing 4-3). A Servlet 3.0+ compatible container scan the classpath for classes that implement the `javax.servlet.ServletContainerInitializer` interface, and it invokes the `onStartup` method on those classes. By adding a `javax.servlet.annotation.HandlesTypes` annotation on these classes, you can also be handed classes that you need to further configure your web application (this is the mechanism that allows the fourth option to use `org.springframework.web.WebApplicationInitializer`).

Like web fragments, `ServletContainerInitializer` allows a modularized configuration of your web application, but now in a Java-based way. Using Java gives you all the added benefits of using the Java language instead of XML. At this point, you have strong typing, can influence the construction of your servlet, and have an easier way of configuring your servlets (in an XML file, this is done by adding `init-param` and/or `context-param` elements in the XML file).

A Java-based Configuration

```
package com.apress.prospringmvc.bookstore.web;
import java.util.Set;
// javax.servlet imports omitted.
import org.springframework.web.servlet.DispatcherServlet;
public class BookstoreServletContainerInitializer
implements ServletContainerInitializer {
    @Override
    public void onStartup(Set<Class<?>> classes, ServletContext
servletContext)
        throws ServletException {
        ServletRegistration.Dynamic registration;
        registration = servletContext.addServlet("ds", DispatcherServlet.
class);
        registration.setLoadOnStartup(1);
        registration.addMapping("/");
    }
}
```

Using WebApplicationInitializer

Now it's time to look at option 4 for configuring your application while using Spring. Spring provides a `ServletContainerInitializer` implementation (`org.springframework.web.SpringServletContainerInitializer`) that makes life a little easier (see Listing 4-4). The implementation provided by the Spring Framework detects and instantiates all instances of `org.springframework.web.WebApplicationInitializer` and calls the `onStartup` method of those

instances.

Listing 4-4. The WebApplicationInitializer Configuration

```
package com.apress.prospringmvc.bookstore.web;
// javax.servlet imports omitted
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.servlet.DispatcherServlet;
public class BookstoreWebApplicationInitializer
implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext servletContext)
    throws ServletException {
        ServletRegistration.Dynamic registration
        registration = servletContext.addServlet("dispatcher",
        DispatcherServlet.class);
        registration.addMapping("/");
        registration.setLoadOnStartup(1);
    }
}
```

Using Spring Boot

When using Spring Boot, you don't need to configure DispatcherServlet manually. Spring Boot automatically configure based on the detected configuration. The properties mentioned in Table 4-2 are mostly configurable through properties in the spring.mvc namespace. See Listing 4-5 for a basic sample.

Listing 4-5. BookstoreApplication Using Spring Boot

```
package com.apress.prospringmvc.bookstore;
@SpringBootApplication
public class BookstoreApplication {
    public static void main(String[] args) {
        SpringApplication.run(BookstoreApplication.class, args);
    }
}
```

A specialized WebApplicationInitializer is needed when using Spring Boot in a classic WAR application. Spring Boot provides the SpringBootServletInitializer for this. See Listing 4-6 for a sample.

Listing 4-6. BookstoreApplication Using Spring Boot in a WAR

```
package com.apress.prospringmvc.bookstore;
@SpringBootApplication
public class BookstoreApplication extends SpringBootServletInitializer {
    public static void main(String[] args) {
        SpringApplication.run(BookstoreApplication.class, args);
    }
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
```

```

builder) {
return builder.sources(BookstoreApplication.class);
}
}

```

Configuring DispatcherServlet

Configuring `org.springframework.web.servlet.DispatcherServlet` is a two-step process. The first step is to configure the servlet's behavior by setting properties directly on the dispatcher servlet (the declaration). The second step is to configure the components in the application context (initialization).

The dispatcher servlet comes with a lot of default settings for components. This saves you from doing a lot of configuration for basic behavior, and you can override and extend the configuration however you want. In addition to the default configuration for the dispatcher servlet, there is also a default for Spring MVC. This can be enabled by using an `org.springframework.web.servlet.config.annotation.EnableWebMvc` annotation (see the "Enabling Features" section in Chapter 2).

DispatcherServlet Properties

The dispatcher servlet has several properties that can be set. All these properties have a setter method, and all can be either set programmatically or by including a servlet initialization parameter. Table 4-2 lists and describes the properties available on the dispatcher servlet.

Property	Default	Description
<code>cleanupAfterInclude</code>	True	Indicates whether to clean up the request attributes after an include request. In general, the default suffices, and this property should only be set to false in special cases.
<code>contextAttribute</code>	Null	Stores the application context for this servlet. It is useful if the application context is created by some means other than the servlet itself.
<code>contextClass</code>	<code>org.springframework.web.context.support.XmlWebApplicationContext</code>	Configures the type of <code>org.springframework.web.context.WebApplicationContext</code> to be constructed by the servlet (it needs a default constructor). Configured using the given <code>contextConfigLocation</code> . It isn't needed if you pass in an application context by using the constructor.
<code>contextConfigLocation</code>	<code>[servlet-name]-servlet.xml</code>	Indicates the location of the configuration files for the specified application context class.
<code>contextId</code>	Null	Provides the application context ID. For example, this is used when the context is logged or sent to <code>System.out</code> .
<code>contextInitializers</code> <code>contextInitializer</code> Classes	Null	Use the optional <code>org.springframework.context.ApplicationContextInitializer</code> classes to do some initialization logic for the application context, such as activating a certain profile.

Property	Default	Description
<code>detectAllHandlerAdapters</code>	<code>True</code>	Detects all <code>org.springframework.web.servlet.HandlerAdapter</code> instances from the application context. When set to false, a single one is detected by using the special name, <code>handlerAdapter</code> .
<code>detectAllHandlerExceptionResolvers</code>	<code>True</code>	Detects all <code>org.springframework.web.servlet.HandlerExceptionResolver</code> instances from the application context. When set to false, a single one is detected by using the special name, <code>handlerExceptionResolver</code> .
<code>detectAllHandlerMappings</code>	<code>True</code>	Detects all <code>org.springframework.web.servlet.HandlerMapping</code> beans from the application context. When set to false, a single one is detected by using the special name, <code>handlerMapping</code> .
<code>detectAllViewResolvers</code>	<code>True</code>	Detects all <code>org.springframework.web.servlet.ViewResolver</code> beans from the application context. When set to false, a single one is detected by using the special name, <code>viewResolver</code> .
<code>dispatchOptionsRequest</code>	<code>False</code>	Indicates whether to handle HTTP OPTIONS requests. The default is false; when set to true, you can also handle HTTP OPTIONS requests.

Property	Default	Description
<code>dispatchTraceRequest</code>	<code>False</code>	Indicates whether to handle HTTP TRACE requests. The default is false; when set to true, you can also handle HTTP TRACE requests.
<code>environment</code>	<code>org.springframework.web.context.support.StandardServletEnvironment</code>	Configures <code>org.springframework.core.env.Environment</code> for this servlet. The environment specifies which profile is active and can hold properties specific to this environment.
<code>namespace</code>	<code>[servletname]-servlet</code>	Use this namespace to configure the application context.
<code>publishContext</code>	<code>True</code>	Indicates whether the servlet's application context is being published to the <code>javax.servlet.ServletContext</code> . For production, we recommend that you set this to false.
<code>publishEvents</code>	<code>True</code>	Indicates whether to fire after request processing <code>org.springframework.web.context.support.ServletRequestHandledEvent</code> . You can use <code>org.springframework.context.ApplicationListener</code> to receive these events.
<code>threadContextInheritable</code>	<code>False</code>	Indicates whether to expose the <code>LocaleContext</code> and <code>RequestAttributes</code> to child threads created from the request handling thread.

The Spring MVC Components

In the previous sections, you learned about the request processing workflow and the components used in it. You also learned how to configure `org.springframework.web.servlet.DispatcherServlet`. In this section, you take a closer look at all the components involved in the request processing workflow. For example, you explore the different components' APIs and see which implementations ship with the Spring

Framework.

HandlerMapping

Handler mapping determines which handler to dispatch the incoming request to. A criterion that you could use to map the incoming request is the URL; however, implementations (see Figure 4-10) are free to choose what criteria to use to determine the mapping.

The API for `org.springframework.web.servlet.HandlerMapping` consists of a single method (see Listing 4-9). This method is called by `DispatcherServlet` to determine `org.springframework.web.servlet.HandlerExecutionChain`. It is possible to have more than one handler mapping configured. The servlet calls the different handler mappings in sequence until one of them doesn't return null.

Listing 4-9. The HandlerMapping API

```
package org.springframework.web.servlet;
import javax.servlet.http.HttpServletRequest;
public interface HandlerMapping {
    HandlerExecutionChain getHandler(HttpServletRequest request)
    throws Exception;
}
```

BeanNameUrlHandlerMapping

The `org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping` implementation is one of the default strategies used by the dispatcher servlet. This implementation treats any bean with a name that starts with a `/` as a potential request handler. A bean can have multiple names, and names can also contain a wildcard, expressed as an `*`.

This implementation uses ant-style regular expressions to match the URL of the incoming request to the name of a bean. It follows this algorithm.

1. Attempt exact match; if found, exit.
2. Search all registered paths for a match; the most specific one wins.
3. If no matches are found, return the handler mapped to `/*` or to the default handler (if configured).

Listing 4-10 shows how to use a bean name and map it to the `/index.htm` URL. In the sample application, you could now use `http://localhost:8080/chapter4-bookstore/index.htm` to call this controller.

Listing 4-10. The BeanNameUrlHandlerMapping sample Configuration

```
package com.apress.prospringmvc.bookstore.web.config;
import java.util.Properties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.apress.prospringmvc.bookstore.web.IndexController;
```

@Configuration

```
public class WebMvcContextConfiguration {  
    @Bean(name = { "/index.htm" })  
    public IndexController indexController() {  
        return new IndexController();  
    }  
}
```

SimpleUrlHandlerMapping

This implementation requires explicit configuration, as opposed to `org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping`, and it doesn't autodetect mappings. Listing 4-11 shows a sample configuration. Again, you map the controller to the `/index.htm`.

Listing 4-11. The SimpleUrlHandlerMapping Sample Configuration

```
package com.apress.prospringmvc.bookstore.web.config;  
// Other imports omitted see Listing 4-10  
import org.springframework.web.servlet.HandlerMapping;  
import org.springframework.web.servlet.handler.SimpleUrlHandlerMapping;  
@Configuration  
public class WebMvcContextConfiguration {  
    @Bean  
    public IndexController indexController() {  
        return new IndexController();  
    }  
    @Bean  
    public HandlerMapping simpleUrlHandlerMapping() {  
        var mappings = new Properties();  
        mappings.put("/index.htm", "indexController");  
        var urlMapping = new SimpleUrlHandlerMapping();  
        urlMapping.setMappings(mappings);  
        return urlMapping;  
    }  
}
```

RequestMappingHandlerMapping

The `RequestMappingHandlerMapping` implementation is more sophisticated. It uses annotations to configure mappings. The annotation can be on either the class and/or the method level. To map the `com.apress.prospringmvc.bookstore.web.IndexController` to `/index.htm`, you need to add the `@RequestMapping` annotation. Listing 4-12 is the controller, and Listing 4-13 shows the sample configuration.

Listing 4-12. The IndexController with RequestMapping

```
package com.apress.prospringmvc.bookstore.web;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;
```

```

import org.springframework.web.servlet.ModelAndView;
@Controller
public class IndexController {
    @RequestMapping(value = "/index.htm")
    public ModelAndView indexPage() {
        return new ModelAndView("/WEB-INF/views/index.jsp");
    }
}

```

Listing 4-13. An annotation-based sample Configuration package com.apress.prospringmvc.bookstore.web.config;
 // Other imports omitted see Listing 4-10

```

@Configuration
public class WebMvcContextConfiguration {
    @Bean
    public IndexController indexController() {
        return new IndexController();
    }
}

```

RouterFunctionMapping

The org.springframework.web.servlet.function.support.HandlerFunctionAdapter implementation is a functional way to define handlers. Listing 4-14 shows the functional style of writing a handler to render the index page.

Listing 4-14. A Functional-Style Sample Configuration package com.apress.prospringmvc.bookstore.web.config;
 // Other imports omitted see Listing 4-10

```

import org.springframework.web.servlet.function.RouterFunction;
import org.springframework.web.servlet.function.ServerRequest;
import org.springframework.web.servlet.function.ServerResponse;
@Configuration
public class WebMvcContextConfiguration {
    @Bean
    public RouterFunction<ServerResponse> routes() {
        return route()
            .GET("/", response -> ok().render("index"))
            .build();
    }
}

```

HandlerAdapter

org.springframework.web.servlet.HandlerAdapter is the glue between the dispatcher servlet and the selected handler. It removes the actual execution logic from the dispatcher servlet, which makes the dispatcher servlet infinitely extensible. Consider this component the glue between the servlet and the actual handler implementation.

Listing 4-15 shows the HandlerAdapter API.

The HandlerAdapter API

```
package org.springframework.web.servlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public interface HandlerAdapter {
    boolean supports(Object handler);
    ModelAndView handle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception;
    long getLastModified(HttpServletRequest request, Object handler);
}
```

HttpRequestHandlerAdapter

org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter knows how to execute org.springframework.web.servlet.mvc.HttpRequestHandler instances. This handler adapter is mostly used by Spring Remoting to support some of the HTTP remoting options. However, there are two implementations of the org.springframework.web.servlet.mvc.HttpRequestHandler interface that you also use. One serves static resources, and the other forwards incoming requests to the servlet container's default servlet (see Chapter 5 for more information).

SimpleControllerHandlerAdapter

org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter knows how to execute org.springframework.web.servlet.mvc.Controller implementations. It returns org.springframework.web.servlet.ModelAndView from the handleRequest method of the controller instance.

SimpleServletHandlerAdapter

It can be convenient to configure javax.servlet.Servlet instances in the application context and put them behind the dispatcher servlet. To execute those servlets, you need org.springframework.web.servlet.handler.SimpleServletHandlerAdapter. It knows how to execute javax.servlet.Servlet, and it always returns null because it expects the servlet to handle the response itself.

MultipartResolver

The org.springframework.web.multipart.MultipartResolver strategy interface determines whether an incoming request is a multipart file request (for file uploads), and if so, it wraps the incoming request in org.springframework.web.multipart.MultipartHttpServletRequest. The wrapped request can then get easy access to the underlying multipart files from the form. File uploading is explained in Chapter 7.

Listing 4-16 shows the MultipartResolver API.

Listing 4-16. The MultipartResolver API

```
package org.springframework.web.multipart;
import javax.servlet.http.HttpServletRequest;
public interface MultipartResolver {
    boolean isMultipart(HttpServletRequest request);
}
```

```

MultipartHttpServletRequest resolveMultipart(HttpServletRequest request)
throws MultipartException;
void cleanupMultipart(MultipartHttpServletRequest request);
}

```

LocaleResolver

The `org.springframework.web.servlet.LocaleResolver` strategy interface determines which `java.util.Locale` to render the page. In most cases, it resolves validation messages or labels in the application. The different implementations are shown in Figure 4-14 and described in the following subsections.

The LocaleResolver API

```

package org.springframework.web.servlet;
import java.util.Locale;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public interface LocaleResolver {
    Locale resolveLocale(HttpServletRequest request);
    void setLocale(HttpServletRequest request, HttpServletResponse response,
        Locale locale);
}

```

AcceptHeaderLocaleResolver

The `org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver` implementation simply delegates to the `getLocale` method of the current `javax.servlet.HttpServletRequest`. It uses the `Accept-Language` HTTP header to determine the language. The client sets this header value; this resolver doesn't support changing the locale.

CookieLocaleResolver

The `org.springframework.web.servlet.i18n.CookieLocaleResolver` implementation uses `javax.servlet.http.Cookie` to store the locale to use. This is particularly useful in cases where you want an application to be as stateless as possible. The actual value is stored on the client side, and it is sent to you with each request. This resolver allows the locale to be changed (you can find more information on this in Chapter 6). This resolver also allows you to configure the name of the cookie and a default locale to use. If no value can be determined for the current request (i.e., there is neither a cookie nor a default locale), this resolver falls back to the request's locale (see `AcceptHeaderLocaleResolver`).

FixedLocaleResolver

`org.springframework.web.servlet.i18n.FixedLocaleResolver` is the most basic implementation of `org.springframework.web.servlet.LocaleResolver`. It allows you to configure a locale to use throughout the whole application. This configuration is fixed; as such, it cannot be changed.

SessionLocaleResolver

The `org.springframework.web.servlet.i18n.SessionLocaleResolver` implementation uses the `javax.servlet.http.HttpSession` to store the value of the locale. The name of the attribute, as well as a default locale, can be configured. If no value can be determined for the current request (i.e., there is neither a value stored in the session nor a default locale), then it falls back to the request's locale (see `AcceptHeaderLocaleResolver`). This resolver also lets you change the locale (see Chapter 6 for more information).

ThemeResolver

The `org.springframework.web.servlet.ThemeResolver` strategy interface determines which theme to render the page. There are several implementations; these are shown in Figure 4-15 and explained in the following subsections. How to apply theming is explained in Chapter 8. If no theme name can be resolved, then this resolver uses the hardcoded default theme.

Listing 4-18. The ThemeResolver API

```
package org.springframework.web.servlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public interface ThemeResolver {
    String resolveThemeName(HttpServletRequest request);
    void setThemeName(HttpServletRequest request, HttpServletResponse
response, String themeName);
}
```

CookieThemeResolver

`org.springframework.web.servlet.theme.CookieThemeResolver` uses `javax.servlet.http.Cookie` to store the theme to use. This is particularly useful where you want your application to be as stateless as possible. The actual value is stored on the client side and sent to you with each request. This resolver allows the theme to be changed; you can find more information on this in Chapters 6 and 8. This resolver also allows you to configure the name of the cookie and a theme locale to use.

FixedThemeResolver

`org.springframework.web.servlet.theme.FixedThemeResolver` is the most basic implementation of `org.springframework.web.servlet.ThemeResolver`. It allows you to configure a theme to use throughout the whole application. This configuration is fixed; as such, it cannot be changed.

SessionThemeResolver

`org.springframework.web.servlet.theme.SessionThemeResolver` uses `javax.servlet.http.HttpSession` to store the value of the theme. The name of the attribute, as well as a default theme, can be configured.

HandlerExceptionResolver

In most cases, you want to control how you handle an exception that occurs during the handling of a request. You can use a `HandlerExceptionResolver` for this. The API (see Listing 4-19) consists of a single method that is called on the `org.springframework.web`.

servlet.HandlerExceptionResolvers detected by the dispatcher servlet. The resolver can choose to handle the exception itself or to return an org.springframework.web.servlet.ModelAndView implementation that contains a view to render and a model (generally containing the exception thrown).

Listing 4-19. The HandlerExceptionResolver API

```
package org.springframework.web.servlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public interface HandlerExceptionResolver {
    ModelAndView resolveException(HttpServletRequest request,
    HttpServletResponse response, Object
    handler, Exception ex);
}
```

Introducing Controllers

The controller is the component that is responsible for responding to the action the user takes. This action could be a form submission, clicking a link, or simply accessing a page. The controller selects or updates the data needed for the view. It also selects the name of the view to render or can render the view itself. With Spring MVC, we have two options when writing controllers. We can either implement an interface or put an annotation on the class.

Interface-based Controllers

To write an interface-based controller, we need to create a class that implements the org.springframework.web.servlet.mvc.Controller interface. Listing 5-1 shows the API for that interface. When implementing this interface, we must implement the handleRequest method. This method needs to return an org.springframework.web.servlet.ModelAndView object or null when the controller handles the response itself.

Listing 5-1. The Controller Interface

```
package org.springframework.web.servlet.mvc;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
@FunctionalInterface
public interface Controller {
    ModelAndView handleRequest(HttpServletRequest request,
    HttpServletResponse response) throws Exception;
}
```

Annotation-based Controllers

To write an annotation-based controller, we need to write a class and put the org.springframework.stereotype.Controller annotation on that class. Also, we need to add an org.springframework.web.bind.annotation.RequestMapping annotation to the class, a method, or both. Listing 5-3 shows an annotation-based approach to our

IndexController.

An Annotation-based IndexController

```
package com.apress.prospringmvc.bookstore.web;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
@Controller
public class IndexController {
    @RequestMapping(value = "/index.htm")
    public ModelAndView indexPage() {
        return new ModelAndView("index");
    }
}
```

Configuring View Controllers

The two controller samples we have written so far are called view controllers. They don't select data; rather, they only select the view name to render. If we had a large application with more of these views, it would become cumbersome to maintain and write these. Spring MVC can help us here. Enabling us simply to add `org.springframework.web.servlet.mvc.ParameterizableViewController` to our configuration and to configure it accordingly. We would need to configure an instance to return index as a view name and map it to the `/index.htm` URL. Listing 5-4 shows what needs to be added to make this work.

A ParameterizableViewController Configuration

```
package com.apress.prospringmvc.bookstore.web.config;
import org.springframework.web.servlet.mvc.ParameterizableViewController;
// Other imports omitted
@Configuration
public class WebMvcContextConfiguration implements WebMvcConfigurer {
    // Other methods omitted
    @Bean(name = "/index.htm")
    public Controller index() {
        ParameterizableViewController index = new
        ParameterizableViewController();
        index.setViewName("index");
        return index;
    }
}
```

Request-Handling Methods

Class	Method	Description
	<code>@RequestMapping(value="/order.htm")</code>	Maps to all requests on the order.htm URL
<code>@RequestMapping("/order.htm")</code>	<code>@RequestMapping(method=RequestMethod.GET)</code>	Maps to all GET requests to the order.html URL
<code>@RequestMapping("/order.*")</code>	<code>@RequestMapping(method={RequestMethod.PUT, RequestMethod.POST})</code>	Maps to all PUT and POST requests to the order.* URL. * means any suffix or extension such as .htm, .doc, .xls, and so on
<code>@RequestMapping(value="/customer.htm", consumes="application/json")</code>	<code>@RequestMapping(produces="application/xml")</code>	Maps to all requests that post JSON and accept XML as a response
<code>@RequestMapping(value="/order.htm")</code>	<code>@RequestMapping(params="add-line", method=RequestMethod.POST)</code>	Maps to all POST requests to the order.htm URL that include an add-line parameter
<code>@RequestMapping(value="/order.htm")</code>	<code>@RequestMapping(headers="!VIA")</code>	Maps to all requests to the order.htm URL that don't include a VIA HTTP header

Supported Method Argument Annotations

In addition to explicitly supported types (as mentioned in the previous section), there are also a couple of annotations that we can use to annotate our method arguments (see Table 5-5). Some of these can also be used with the method argument types mentioned in Table 5-4. In that case, they specify the name of the attribute in the request, cookie, header, or response, as well as whether the parameter is required.

Table 5-5. The Supported Method Argument Annotations

Argument Type	Description
<code>RequestParam</code>	Binds the argument to a single request parameter or all request parameters.
<code>RequestHeader</code>	Binds the argument to a single request header or all request headers. ⁵
<code>RequestBody</code>	Gets the request body for arguments with this annotation. The value is converted using <code>org.springframework.http.converter.HttpMessageConverter</code> .
<code>RequestPart</code>	Binds the argument to the part of a multipart form submission.
<code>ModelAttribute</code>	Binds and validates arguments with this annotation. The parameters from the incoming request are bound to the given object.
<code>PathVariable</code>	Binds the method parameter to a path variable specified in the URL mapping (the value attribute of the <code>RequestMapping</code> annotation).
<code>CookieValue</code>	Binds the method parameter to a <code>javax.servlet.http.Cookie</code> .
<code>SessionAttribute</code>	Binds the method parameter to a session attribute.
<code>RequestAttribute</code>	Binds the method parameter to a request attribute (not to be confused with a request parameter).
<code>MatrixVariable</code>	Binds the method parameter to a name-value pair within a path-segment.

```
The HandlerMethodArgumentResolver Interface
package org.springframework.web.method.support;
import org.springframework.core.MethodParameter;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.support.WebDataBinderFactory;
import org.springframework.web.context.request.NativeWebRequest;
public interface HandlerMethodArgumentResolver {
    boolean supportsParameter(MethodParameter parameter);
    Object resolveArgument(MethodParameter parameter,
        ModelAndViewContainer mavContainer,
        NativeWebRequest webRequest,
        WebDataBinderFactory binderFactory)
    throws Exception;
}
```

Oracle provides 23 different SQL datatypes. Briefly, they are as follows:

- **CHAR**: A fixed-length character string that will be blank padded with spaces to its maximum length. A non-null `CHAR(10)` will always contain 10 bytes of information using the default National Language Support (NLS) settings. We will cover NLS implications in more detail shortly. A `CHAR` field may store up to 2000 bytes of information.
- **NCHAR**: A fixed-length character string that contains UNICODE

formatted data. Unicode is a character-encoding standard developed by the Unicode Consortium with the aim of providing a universal way of encoding characters of any language, regardless of the computer system or platform being used. The NCHAR type allows a database to contain data in two different character sets: the CHAR type and NCHAR type use the database's character set and the national character set, respectively. A non-null NCHAR(10) will always contain 10 characters of information (note that it differs from the CHAR type in this respect). An NCHAR field may store up to 2000 bytes of information.

- **VARCHAR2**: Also currently synonymous with VARCHAR. This is a variable-length character string that differs from the CHAR type in that it is not blank padded to its maximum length. A VARCHAR2(10) may contain between 0 and 10 bytes of information using the default NLS settings. A VARCHAR2 may store up to 4000 bytes of information. Starting with Oracle 12c, a VARCHAR2 can be configured to store up to 32,767 bytes of information (see the “Extended Datatypes” section in this chapter for further details).
- **NVARCHAR2**: A variable length character string that contains UNICODE formatted data. An NVARCHAR2(10) may contain between 0 and 10 characters of information. An NVARCHAR2 may store up to 4,000 bytes of information. Starting with Oracle 12c, an NVARCHAR2 can be configured to store up to 32,767 bytes of information (see the “Extended Datatypes” section in this chapter for further details).
- **RAW**: A variable-length binary datatype, meaning that no character set conversion will take place on data stored in this datatype. It is considered a string of binary bytes of information that will simply be stored by the database. A RAW may store up to 2000 bytes of information. Starting with Oracle 12c, a RAW can be configured to store up to 32,767 bytes of information (see the “Extended Datatypes” section in this chapter for further details).
- **NUMBER**: This datatype is capable of storing numbers with up to 38 digits of precision. These numbers may vary between 1.0×10^{-130} and up to but not including 1.0×10^{126} . Each number is stored in a variable-length field that varies between 0 bytes (for NULL) and 22 bytes. Oracle NUMBER types are very precise—much more so than normal FLOAT and DOUBLE types found in many programming languages.
- **BINARY_FLOAT**: This is a 32-bit single-precision floating-point number. It can support at least six digits of precision and will consume 5 bytes of storage on disk.
- **BINARY_DOUBLE**: This is a 64-bit double-precision floating-point number. It can support at least 15 digits of precision and will consume 9 bytes of storage on disk.
- **LONG**: This type is capable of storing up to 2GB of character data (2

gigabytes, not characters, as each character may take multiple bytes in a multibyte character set). LONG types have many restrictions (I'll discuss later) that are provided for backward compatibility, so it is strongly recommended you do not use this type in new applications. When possible, convert from LONG to CLOB types in existing applications.

- **LONG RAW:** The LONG RAW type is capable of storing up to 2GB of binary information. For the same reasons as noted for LONGs, it is recommended you use the BLOB type in all future development and, when possible, in existing applications as well.
- **DATE:** This is a fixed-width 7-byte date/time datatype. It will always contain the seven attributes of the century, the year within the century, the month, the day of the month, the hour, the minute, and the second.
- **TIMESTAMP:** This is a fixed-width 7- or 11-byte date/time datatype (depending on the precision). It differs from the DATE datatype in that it may contain fractional seconds; up to nine digits to the right of the decimal point may be preserved for TIMESTAMPS with fractional seconds.
- **TIMESTAMP WITH TIME ZONE:** This is a fixed-width 13-byte date/time datatype, but it also provides for TIME ZONE support. Additional information regarding the time zone is stored with the TIMESTAMP in the data, so the TIME ZONE originally inserted is preserved with the data.
- **TIMESTAMP WITH LOCAL TIME ZONE:** This is a fixed-width 7- or 11-byte date/time datatype (depending on the precision), similar to the TIMESTAMP; however, it is time zone sensitive. Upon modification in the database, the TIME ZONE supplied with the data is consulted, and the date/time component is normalized to the local database time zone. So, if you were to insert a date/time using the time zone US/Pacific and the database time zone was US/Eastern, the final date/time information would be converted to the Eastern time zone and stored as a TIMESTAMP. Upon retrieval, the TIMESTAMP stored in the database would be converted to the time in the session's time zone.
- **INTERVAL YEAR TO MONTH:** This is a fixed-width 5-byte datatype that stores a duration of time, in this case as a number of years and months. You may use intervals in date arithmetic to add or subtract a period of time from a DATE or the TIMESTAMP types.
- **INTERVAL DAY TO SECOND:** This is a fixed-width 11-byte datatype that stores a duration of time, in this case as a number of days and hours, minutes, and seconds, optionally with up to nine digits of fractional seconds.
- **BLOB:** This datatype permits for the storage of up to (4 gigabytes – 1) * (database block size) bytes of data in Oracle. BLOBs contain “binary” information that is not subject to character set conversion. This

would be an appropriate type in which to store a spreadsheet, a word processing document, image files, and the like.

- CLOB: This datatype permits for the storage of up to (4 gigabytes – 1) * (database block size) bytes of data in Oracle. CLOBs contain information that is subject to character set conversion. This would be an appropriate type in which to store large plain text information. Note that I said large plain text information; this datatype would not be appropriate if your plain text data is 4000 bytes or less—for that you would want to use the VARCHAR2 datatype.
- NCLOB: This datatype permits for the storage of up to (4 gigabytes – 1) * (database block size) bytes of data in Oracle. NCLOBs store information encoded in the national character set of the database and are subject to character set conversions just as CLOBs are.
- BFILE: This datatype permits you to store an Oracle directory object (a pointer to an operating system directory) and a file name in a database column and to read this file. This effectively allows you to access operating system files available on the database server in a read-only fashion, as if they were stored in the database table itself.
- ROWID: A ROWID is effectively a 10-byte address of a row in a database. Sufficient information is encoded in the ROWID to locate the row on disk, as well as identify the object the ROWID points to (the table and so on).
- UROWID: A UROWID is a universal ROWID and is used for tables—such as IOTs and tables accessed via gateways to heterogeneous databases—that do not have fixed ROWIDs. The UROWID is a representation of the primary key value of the row and hence will vary in size depending on the object to which it points.
- JSON: New with Oracle 21c is the JSON datatype. You can now store JSON data natively in the database in a binary format.

Одна или несколько колонок в таблице могут быть обозначены как первичный ключ.

Первичным ключом обозначаются колонки таблицы, содержащие набор уникальных значений, по которым мы можем однозначно идентифицировать строку в рамках этой таблицы.

Первичный ключ не может содержать пустые значения, так как всегда имеет ограничение NOT NULL.

Вторичный ключ – так обозначается колонка таблицы, в которой есть данные, используемые для связи с другой таблицей.

В СУБД есть понятие схемы – это особая логическая область, ассоциированная с заданной учетной записью, которая объединяет несколько объектов базы данных.

```
CREATE TABLE kuku(  
Id NUMBER PRIMARY KEY,  
Somestring VARCHAR2(300),  
Somenum NUMBER  
)
```

ALTER TABLE - изменить таблицу
DROP COLUMN. - удалить столбец
ALTER TABLE TABLE_NAME DROP COLUMN column_NAME;

Синтаксис изменения типа колонки:
ALTER TABLE TABLE_NAME MODIFY (column_NAME
DATA_type);

Синтаксис команды SQL для удаления таблицы:
DROP TABLE TABLE_NAME;

удалить таблицу BILLING_PERIOD со связанными данными в таблице PERIODS.
DROP TABLE BILLING_PERIODS CASCADE;

Важные замечания

Первичный ключ может состоять из одной или нескольких колонок.

Пример:

```
ALTER TABLE TABLE_NAME ADD CONSTRAINT  
constraiNt_NAME PRIMARY KEY (column1);
```

или же

```
ALTER TABLE TABLE_NAME ADD CONSTRAINT  
constraiNt_NAME PRIMARY KEY (column1, columnN);
```

Для колонок таблицы в базе данных можно создавать ограничения.

Ограничения – это специальные синтаксические конструкции уровня колонок таблицы, которые предназначены для поддержания ссылочной целостности данных или для вставки пра-

вильных данных согласно бизнес-логике приложения.

То есть ограничения допускают вставку в ячейку таблицы только определенных данных, ограниченных заданными правилами.

NOT NULL

UNIQUE - должен быть уникальным

Смысл данного ограничения в том, что в колонке некоторой таблицы (вторичный ключ) могут находиться только значения, которые есть в другой, основной таблице в колонке

первичного ключа.

Синтаксис

```
ALTER TABLE for_TABLE  
ADD CONSTRAINT fk_const_NAME  
FOREIGN KEY (fk_column)  
REFERENCES primary_table (pk_column);
```

Ограничение CHECK на вставку и изменение данных

– вычитание;

Синтаксис

```
ALTER TABLE TABLENAME ADD CONSTRAINT CHECK_NAME  
CHECK (condition);
```

Здесь condition – условие, CHECK_NAME – наименование ограничения, TABLENAME – имя таблицы.

Пример

Ограничение в таблице MAN на возраст (YEAROLD) больше 16 лет:

```
ALTER TABLE MAN ADD CONSTRAINT  
CHECK_YEAROLD_MAN  
CHECK (YEAROLD > 16);
```

Ограничение уникальности можно также создавать для нескольких колонок таблицы, это делается следующим образом:

```
ALTER TABLE CITY ADD CONSTRAINT CITY_uniq  
UNIQUE (CITYNAME, CITYCODE);
```

При этом отдельно необходимо контролировать вставку пустых значений для соответствующих полей таблицы.

Существуют дополнительные опции для создания ограничений ссылочной целостности:

- ON DELETE CASCADE – автоматическое удаление связанных строк по внешнему ключу;
- ON DELETE NULL – значение внешнего ключа устанавливается в NULL.

Индексы – это специальные ссылочные массивы в базах данных. Назначение индексов – ускорение поиска данных, процессов сортировки данных. Обычно индексы увеличивают производительность запросов к базе данных.

Индексы создаются для определенной колонки (колонок) таблицы.

Процесс пересоздания индексов может занимать значительное время, это необходимо учитывать при операциях вставки и обновления, удаления данных.

Синтаксис

```
CREATE INDEX idx_NAME ON TABLE_NAME (column_NAME);
```

idx_NAME – наименование индекса;

TABLE_NAME – наименование таблицы, где создается индекс;

column_NAME – наименование колонки, для которой создается индекс.

Пример создания индекса для колонки MARK таблицы AUTO:

```
CREATE INDEX idx_AUTO_MARK ON AUTO (MARK);
```

Реверсивный индекс

Если нам необходимо более часто читать записи, отсортированные в обратном порядке,

тогда имеет смысл использовать реверсивные индексы. Например, есть таблица валют, в своих

расчетах мы чаще используем данные с более поздней датой курса валют, в этом случае дей-

ствительно лучше использовать реверсивный индекс для даты курса валют.

Синтаксис

```
CREATE INDEX IDx_NAME ON TABLE_NAME (column_NAME)
REVERSE;
```

Пример: создание реверсивного индекса для колонки MARK таблицы AUTO.

```
CREATE INDEX reg_DATE ON AUTO (reg_num) REVERSE;
```

Удаление индекса

Для удаления индекса используется команда

```
DROP INDEX IDx_NAME;
```

Обычно использование индексов улучшает производительность базы данных, но в таблицах, где предполагается большое количество операций вставок, обновлений, много индексов

использовать не рекомендуется. В этом случае производительность базы данных может суще-

ственно снизиться.

Индексы рекомендуется создавать на колонках, которые используются в операциях объединения.

Индекс автоматически создается для столбцов первичных ключей и для столбцов, на которых есть ограничение уникальности.

При наименовании индексов следует придерживаться следующего правила: IDx_имя таблицы_имена_колонок.

```
SELECT * FROM TABLE_NAME
SELECT CITYCODE, CITYNAME FROM CITY;
SELECT TABLE_NAME.* FROM TABLE_NAME
SELECT TABLE_NAME. column_NAME1, TABLE_NAME.
column_NAME1, TABLE_NAME. column_NAMEn FROM TABLE_NAME
```

Фильтр строк WHERE в запросе SELECT

```
SELECT * FROM CITY WHERE PEOPLES = 300000
```

Операнды сравнения

> больше

<меньше

= строгое равенство

или неравенство! =

Итак, логические операнды позволяют объединять несколько условий, чтобы создать

более сложные критерии выбора строк в операторе WHERE. Разберемся поподробнее, как это работает.

усл1 AND усл2 – логическое И, позволяет объединить несколько условных выражений, так что запрос вернет строку таблицы, если каждое из условий будет верным.

усл1 OR усл2 – логическое Или, позволяет выбрать строки, если одно из заданных условий верно.

NOT усл – логическое отрицание, выбирает строки, если выражение полностью неверно. AND OR и NOT – как указано выше, можно гармонично сочетать в запросе.

Данные, выводимые с помощью запроса SELECT, часто необходимо сортировать по одному или нескольким выводимым колонкам.

Для этого в языке SQL есть специальный оператор ORDER BY, который используется в команде SELECT и пишется в конце запроса. Для того чтобы изменить порядок сортировки на обратный, используется ключевое слово DESC.

ORDER BY используется для сортировки выводимых данных по заданным колонкам: от большего к меньшему, и наоборот.

Текстовые данные будут отсортированы от А до Я или же от Я до А – при использовании ключевого операнда DESC.

Числовые данные сортируются от меньшего числа к большему числу и от большего числа к меньшему при использовании ключевого операнда DESC.

Ограничение на количество выбранных строк ROWNUM, TOP (n)

Для решения этой задачи в разных диалектах языка SQL используются разные синтаксические конструкции: в MS SQL это конструкция TOP, в ORACLE есть специальный предикат ROWNUM, в PostgreSQL, MYSQL для этого существует конструкция LIMIT.

```
SELECT * FROM (SELECT * FROM Auto ORDER BY releasedt) WHERE ROWNUM<кол строк +1
```

MY SQL PostgreSQL

```
SELECT * FROM City WHERE peoples>3000 limit 5
```

Вставка данных в таблицу – INSERT

```
INSERT INTO имя таблицы (колонки через запятую)  
VALUES (значения через запятую);
```

```
INSERT INTO man(phonenum, firstname, lastname, citycode, yearold)
VALUES('120120120','Максим','Леонидов',2,25);
Commit;
```

Команда INSERT является командой модификации данных, поэтому ее выполнение необходимо завершить одной из следующих команд:

COMMIT – фиксация изменений;

ROLLBACK – откат изменений.

Только после выполнения фиксации изменений данные появятся в базе.

Обновление данных – UPDATE

```
UPDATE table_name SET column1 = знач1, column 2 = знач2 , column n = значn WHERE
условия отбора строк для обновления
```

```
UPDATE city SET cityname = cityname || '#' WHERE citycode<2
```

Команда обновления данных UPDATE тоже должна завершаться выполнением COMMIT – фиксацией изменений, либо ROLLBACK – откатом изменений.

Удаление данных – DELETE

```
DELETE city1 WHERE citycode = 7 or citycode =9
```

Команда DELETE также должна завершаться инструкцией COMMIT для фиксации изменений в базе.

Как еще можно очистить таблицу?

Для этого есть специальный оператор TRUNCATE TABLE TABLENAME.

Сложный запрос

```
SELECT MARK as mr, COLOR as cl FROM AUTO WHERE color='СИНИЙ'
```

BETWEEN

В языке SQL есть специальная конструкция, которая позволяет работать с интервалами – своего рода фильтр, позволяющий выбирать данные, соответствующие заданному интер-

валу значений. Этот оператор называется BETWEEN и может использоваться как в выборке SELECT, так и в операциях модификации и удаления данных (UPDATE, DELETE).

```
SELECT перечень полей или * FROM TAB1  
WHERE поле1 BETWEEN нижняя граница интервала AND верхняя граница интервала  
Из TAB1 Будут выбраны строки, соответствующее интервалу ЗАДАННОМУ В BETWEEN
```

```
SELECT * FROM man WHERE yearold BETWEEN 25 and 32
```

DISTINCT, дубликаты значений

```
SELECT DISTINCT перечень полей или * FROM таблица WHERE условия
```

Rownum – это псевдостолбец, значения которого можно увидеть, включив его в любой результирующий набор, например в список столбцов оператора SELECT. Значениям столбца **ROWNUM** присваиваются номера 1, 2, 3, 4, ... N, где N – число строк результирующего набора запроса.

Математика в запросах

Для создания математических выражений в языке SQL используются следующие операции:

- + сложение;
- вычитание;
- / деление;
- * умножение.

А также знакомые нам со школы функции:

SQRT – квадратный корень;

MOD – остаток от деления;

TRUNC – округление до целого;

SIN – синус;

COS – косинус

(на самом деле этих функций больше, мы рассматриваем основные).

```
SELECT 1000*(123-11)/100 as v FROM dual
```

Пустые значения
в базе. NULL, NOT NULL, NVL

```
SELECT * FROM table1 WHERE колонка1 IS NOT NULL
```

Так же в ORACLE SQL применяется специальная встроенная функция NVL, которая преобразует пустое значение в другое, заданное значение.

```
SELECT перечень полей или *, NVL(поле где Null, новое значение) FROM имя таблицы  
WHERE условия выбора строк  
SELECT перечень полей или * новое значение) FROM имя таблицы WHERE NVL(поле где  
Null, значение) = условие сравнения + другие условия выбора строк
```

```
SELECT auto.* FROM auto WHERE NVL(auto.phonenum, 0) != 0
```

Оператор LIKE

```
SELECT перечень полей или * FROM имя таблицы  
WHERE колонка LIKE 'шаблон'
```

Примеры шаблонов LIKE

«%им» – выведет все записи, где значение в заданной колонке заканчивается «им», то есть любое количество символов (%), затем «им»;
«_осква» – выведет все записи, где значение в заданной колонке заканчивается «осква»;
«%лек%» – выведет все записи, где значение в заданной колонке содержит слог «лег»;
«Москв%» – выведет все записи начинающиеся Москв – то есть Москв, а затем любые символы.

Следует помнить, что LIKE применяется только для текстовых данных в ячейке таблицы,
а также то, что шаблон указывается в одинарных кавычках.
Кроме LIKE допустимо использовать конструкцию NOT LIKE.

Функция SYSDATE возвращает текущие дату и время

```
SELECT sysdate FROM DUAL; -- выведет на экран текущее число, и текущее время.
```

Функция TRUNC обрезает время из даты, оставляя только дату.

```
SELECT sysdate FROM dual; -- выведет дату и время текущее  
SELECT trunc(sysdate) FROM dual; -- Только текущее число месяц год
```

```
SELECT TRUNC(sysdate, 'MONTH') "FIRSTDAYOFMONTH" FROM DUAL;
```

Функции и операторы для работы со строками и текстом

Объединение строк – по-правильному конкатенация.

Для объединения строк в языке SQL диалекта ORACLE используется специальная синтаксическая конструкция ||.

```
select 'теле' || 'визор' as q from DUAL  
- телевизор  
select 'LA' || 'DA' as q from DUAL  
- LADA  
select firstname || ' ' || lastname as q from man
```

INSTR – поиск позиции подстроки в строке.

INSTR (STR1, STR2, POSn, DIRECTION) – возвращает позицию STR2 в строке STR1,

где поиск осуществляется в позиции POSn

в направлении DIRECTION 1 – от начала строки, 0 – от окончания строки, то есть откуда мы начинаем поиск – от начала строки или с конца строки.

```
SELECT INSTR('AAABAAAAABA','AB',1) FROM DUAL
-- 3
SELECT INSTR('AAABAAAAABA','AB',5) FROM DUAL
-- 9
SELECT INSTR('AAABAAAAABA','AB',1,1) FROM DUAL
-- 3
```

Length – длина строки в символах.

LENGTH (str1) возвращает длину строки str1 в символах.

Примеры Длина строки «AAA»

```
select length('AAA') as ln from dual
```

Выбор подстроки из строки SUBSTR

SUBSTR (STR1, POS, LEN) выбирает LEN символов в строке str1, начиная с позиции POS.

STR1 – оригинальная строка.

POS – позиция, с которой начинается выделение.

NEWSUB – подстрока, на которую заменяем по умолчанию.

```
SELECT SUBSTR('ABCDEF',2,3) FROM DUAL
```

Замена подстроки в строке REPLACE

REPLACE (SRCSTR, OLDSUB, NEWSUB) – функция, которая возвращает преобразованную строку SRCSTR, где подстрока OLDSUB из строки SRCSTR заменяется на подстроку NEWSUB.

SRCSTR – оригинальная строка.

OLDSUB – заменяемая подстрока.

NEWSUB – подстрока, на которую заменяем, по умолчанию NULL.

Заменить в имени в таблице MAN все буквы а на #.

```
select firstname, replace(firstname,'a','#') as fn from man
```

Математика и пустые значения в запросах. Случайность – RANDOM

```
DBMS_RANDOM.VALUE( low IN NUMBER, high IN NUMBER) RETURN NUMBER;
```

LOW – наименьшее количество в диапазоне для генерации случайного числа.

Номер,

который генерируется, может быть равен LOW;

HIGH – наибольшее число для генерации случайного числа. Номер, который генериру-

ется, будет меньше, чем HIGH.

Возвращаемое значение – NUMBER.

Существует еще один способ генерации случайных текстовых данных с использованием

пакета DBMS_RANDOM.

```
DBMS_RANDOM.STRING opt IN CHAR,  
len IN NUMBER) RETURN VARCHAR2;
```

Параметры

«U», «U» – результат прописные буквы;

«L», «L» – результат строчные буквы;

«A», «A» – результат смешанные буквы, дело;

«X», «X» – результат верхний регистр букв и цифр;

«P», «P» – результат любых печатных символов;

len – длина возвращаемой строки.

```
select DBMS_RANDOM.STRING('u', 4) from dual;  
-- NXBM  
select DBMS_RANDOM.STRING('i', 5) from dual;  
-- TTULB  
select DBMS_RANDOM.STRING('a', 6) from dual;  
-- drpGPp  
select DBMS_RANDOM.STRING('x', 7) from dual;  
-- RXP5CGQ
```

Оператор IN

Для удобной фильтрации выборки по списку значений в SQL существует специальный оператор IN. Он позволяет сравнить значение заданного поля со списком значений и выбирать данные по результатам сравнения.


```
SELECT перечень полей или * FROM таблица WHERE  
поле IN (значение1, значение2, значение )
```

```
SELECT * FROM auto WHERE color in ('СИНИЙ','КРАСНЫЙ','ЗЕЛЕНый')
```

Объединение нескольких таблиц в запросе

```
select перечень полей или * from табл1, табл2  
where табл1.kod = табл2.kod
```

```
SELECT m.firstname, m.lastname, c.cityname,c.peoples FROM man m, city c WHERE c.citycode  
= m.citycode
```

С JOIN ... ON

```
SELECT * FROM man m INNER JOIN city c ON c.citycode = m.citycode
```

Синтаксис LEFT JOIN

SELECT – перечень полей или * FROM – таблица, из которой мы извлекаем все записи;

LEFT JOIN – таблица, где мы извлекаем только совпадающие записи; on – условие объединения

ON (т1.код=т2.код).

Синтаксис RIGHT JOIN SELECT перечень полей или * FROM – мы извлекаем только совпадающие записи;

RIGHT JOIN – таблица, из которой мы извлекаем все записи ON (т1.код=т2.код).

```
SELECT * FROM man m LEFT JOIN auto a on m.phonenum = a.phonenum
```

```
SELECT * FROM man m RIGHT JOIN city c on m.citycode = c.citycode
```

Объединение нескольких таблиц, дополнительные условия и сортировка результатов

Группировка данных и агрегатные функции

Язык SQL позволяет делить данные, полученные в результате выборки, на группы, объединенные по набору колонок – признаков группы.

Например, мы видим, что в запросе по таблице AUTO есть автомобили КРАСНОГО, ЗЕЛЕННОГО и СИНЕГО цветов.

То есть в данной таблице можно определить группы КРАСНЫХ, ЗЕЛЕННЫХ, СИНИХ автомобилей. Мы сгруппировали автомобили по признаку цвета – колонке таблицы COLOR, и мы также можем сгруппировать авто по марке LADA или BMW, колонка MARK.

```
SELECT перечень колонок FROM имя таблицы WHERE условие отбора строк GROUP BY  
колонка группировки 1 , колонка группировки n
```

```
SELECT mark FROM auto GROUP BY mark
```

Агрегатные функции

Определение группировки было бы неполным без понимания агрегатных функций.

Агрегатная функция позволяет нам собрать статистическую информацию по заданной

группе, количество элементов, сумму, среднее значение элементов в группе.

Основные агрегатные функции:

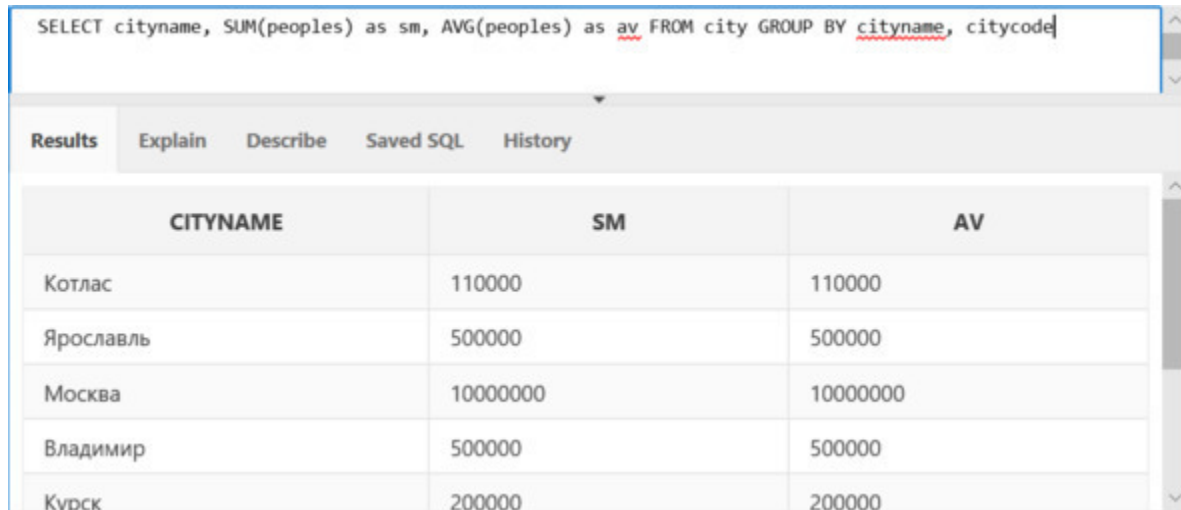
- COUNT (колонка) – возвращает количество элементов в группе;
- SUM () – сумма по заданным значениям, только для числовых данных;
- avg () – среднее по заданным значениям в группе, только для числовых данных;
- MAX () – максимальное значение в группе, только для числовых данных и дат;
- MIN () – минимальное значение в группе, только для числовых данных и дат.

Примеры

Вывести наименование города (CITYNAME), сумму и среднее население (PEOPLES) в группах городов CITY.

```
SELECT cityname, SUM(peoples) as sm, AVG(peoples) as av FROM city GROUP BY cityname, citycode
```

Итог



The screenshot shows a SQL query editor with a query in the top text area. Below the query area is a tabbed interface with 'Results' selected. The results are displayed in a table with three columns: CITYNAME, SM, and AV. The table contains five rows of data for different cities.

```
SELECT cityname, SUM(peoples) as sm, AVG(peoples) as av FROM city GROUP BY cityname, citycode
```

CITYNAME	SM	AV
Котлас	110000	110000
Ярославль	500000	500000
Москва	10000000	10000000
Владимир	500000	500000
Курск	200000	200000

Важные замечания

Важное условие группировки: перечень колонок в запросе после команды SELECT должен входить в группировку GROUP BY или быть частью агрегатной функции.

Ошибочный запрос:

```
SELECT count(yearold) as cnt, firstname, lastname FROM Man GROUP BY firstname
```

Правильный запрос:

```
SELECT count(yearold), firstname, lastname FROM Man GROUP BY firstname, lastname
```

SELECT count(yearold), firstname, lastname FROM Man GROUP BY <u>firstname</u> , lastname		
Results	Explain	Describe
	Saved SQL	History
COUNT(YEAROLD)	FIRSTNAME	LASTNAME
1	Таня	Иванова
1	Андрей	Николаев
1	Алиса	Никифорова
1	Олег	Денисов
1	Миша	Рогозин

Сложные группировки с объединениями, сортировка результатов

```
select mark, count(man.phonenum) cnt from auto inner join man on man.phonenum =
auto.phonenum group by mark
```

HAVING как фильтр для групп и сложные группировки данных. ROWID – уникальный идентификатор строки. Дубликаты строк

В некоторых задачах необходимо применить фильтр уже для результата группировки

записей. Подобный фильтр применяется, например, при ограничении количества выводимых групп.

```
SELECT перечень полей FROM таблица WHERE условия
GROUP BY перечень полей HAVING условие
```

```
SELECT color FROM auto GROUP BY color HAVING color in ('СИНИЙ', 'КРАСНЫЙ')
```

В ORACLE SQL существует специальный идентификатор для каждой строки таблицы –

это колонка ROWID, уникальный идентификатор строки.

Что такое ROWID? Уникальный идентификатор: состоит из номера объекта (32 бита),

относительного номера файла (10 бит), номера блока (22 бита) и номера строки (16 бит).

Как использовать ROWID? Если нет другого уникального идентификатора, первичного

ключа, то можно опираться на ROWID.

Пример работы с ROWID – находим дубли.

```
select t.rowid, t.nm from t where (t.rowid not in ( select min(t.rowid) from t group by nm ))
```

Важные замечания

- Оператор HAVING нельзя использовать в запросах без GROUP BY.
- Оператор HAVING возможно также применять вместе с сортировкой ORDER BY.

Есть ли другие способы удаления дублей из таблиц?

Да, например с использованием EXIST, с использованием аналитических функций.

Используем EXIST:

```
delete auto a where not exists (  
  select 1 from (select max(rowid) rid from auto group by regnum, mark, color, releasedt,  
    phonenum) ar where a.rowid = ar.rid);
```

Подзапрос для множеств WHERE IN SELECT

Важным свойством использования IN является то, что в качестве источника значений для списка оператора может являться другой подзапрос.

SELECT * или колонки через запятую FROM таблиц WHERE имяколонки IN (перечень значений).

```
SELECT * FROM auto WHERE phonenum IN (SELECT phonenum FROM man WHERE  
  yearold>35)
```

Тип данных сравниваемого списка SELECT должен совпадать с типом данных колонки

сравнения, иначе необходимо прибегнуть к преобразованию типов.

При конструкции IN с подзапросом игнорируются NULL-значения. Для работы с NULL-значениями необходимо использовать функцию преобразования NVL.

Подзапросы EXISTS

подзапрос EXIST явля-

ется предикатом, возвращает нам либо истину, либо ложь, то есть подзапрос с EXIST – это

критерий того, будет на экран выведена данная строка либо нет. Если подзапрос возвращает

хоть одну строку, то внешний запрос выводит данные – конкретную связанную строчку, если

нет, данные не выводятся.

Также подзапрос EXISTS подразумевает объединение, то есть внутренний подзапрос свя-

зывается определенным отношением с внешним запросом.

```
SELECT поля FROM таблица1 t1
WHERE EXISTS (SELECT 1 FROM таблица2 t2 WHERE
t1.key = t2.key
)
```

Также эффективно использовать EXIST с логическим операндом NOT, например, используя EXISTS, вывести на экран все города (*) из таблицы CITY, для которых нет соответствующей записи в таблице MAN, связь по полю CITYCODE.

```
SELECT * FROM city c WHERE NOT EXISTS (SELECT 1 FROM man m WHERE c.citycode =
m.citycode )
```

Подзапрос в качестве колонки запроса не должен возвращать более одной строки, в противном случае возникнет ошибка.

Подзапрос как источник данных после FROM

Предикаты ANY, SOME и ALL

Рассмотрим на этом шаге незаслуженно забытые, нечасто используемые на практике кон-

струкции языка SQL – ANY, ALL. Тем не менее данные конструкции обладают

исключитель-

ными возможностями.

Синтаксическая конструкция ANY – это предикат, который является верным, если каж-

дое из всех значений, выведенных подзапросом, удовлетворяет условию для текущей строки

внешнего запроса.

Давайте рассмотрим на примерах:

Вывести из таблицы AUTO только те машины, для которых в таблице AUTO есть

машины такой же марки.

```
select * from auto a where mark = any (select mark from auto a1 where a1.regnum<>a.regnum )
```

```
select * from auto a where mark = any (select mark from auto a1 where a1.regnum<>a.regnum )
```

REGNUM	MARK	COLOR	RELEASEDT	PHONENUM
111126	LADA	ЗЕЛЕНый	01/01/2005	-
111114	LADA	КРАСНый	01/01/2008	9152222221
111122	AUDI	СИНИй	01/01/2011	9213333336
111121	AUDI	СИНИй	01/01/2009	9173333332
111119	LADA	СИНИй	01/01/2017	9213333331

Преобразование типов данных

Есть универсальная функция для преобразования различных типов данных к текстовому формату.

Универсальная функция TO_CHAR.

Рассмотрим несколько примеров работы функции TO_CHAR.

Синтаксис

```
SELECT перечень полей, to_char(поле, 'формат') FROM имя таблицы WHERE условия отбора строк
```

```
select to_char(sysdate, 'dy') as dn from dual
```

Конструкция CAST

Весьма полезной может быть функция преобразования типов – CAST.

Она имеет следующий синтаксис:

```
select cast(поле,тип к которому преобраз) from табл where усл
```

Примеры

Привести число 1000 к типу VARCHAR2 (10).

```
select cast(1000 as varchar2(10)) as cs from dual
```

описанные функции преобразования данных весьма специальные и могут применяться лишь в ORACLE SQL

Объединение таблицы с самой же собой

Пожалуй, самый парадоксальный и необычный способ объединения таблиц JOIN заклю-

чается в том, что язык SQL допускает объединение таблицы с самой же собой, в этом случае

для каждой таблицы создается свой псевдоним и обращение происходит по уникальному псевдониму.

Выбрать из таблицы CITY названия городов и популяцию, а также названия городов и популяцию, где код города = код текущего города +1.

```
select c1.citycode, c2.citycode, c2.cityname, c2.peoples from city c1 left join city c2 on  
c1.citycode +1= c2.citycode
```

Операторы для работы

с множествами – UNION, UNION ALL

Операторы языка SQL UNION и UNION ALL позволяют сделать объединение из нескольких запросов SQL специальным образом. Если при объединении JOIN колонки из разных таблиц располагаются горизонтально друг за другом, то данные при объединении UNION выводятся последовательно, как один набор данных под другим.

Итак, UNION – специальный оператор языка SQL для работы с множествами.

UNION объединяет наборы данных – строки из наборов данных, непосредственно одна за другой. Также часто применяется оператор UNION ALL, который работает так же, как и UNION, но в отличие UNION не выводит записи, если они дублируются, то есть в выводе не будет дублей строк. UNION ALL, напротив, дубли не убирает и выводит дублирующиеся строки на экран.


```
select phonenum from auto union select phonenum from man
```

```
select phonenum from auto
union
select phonenum from man
```

Results	Explain	Describe	Saved SQL	History
PHONENUM				
915222221				
915222222				
915333333				
917333332				
917333334				

```
select phonenum from auto union all select phonenum from man
```

```
select phonenum from auto
union
select phonenum from man
```

Results	Explain	Describe	Saved SQL	History
PHONENUM				
915222221				
915222222				
915333333				
917333332				
917333334				

При использовании операторов UNION есть несколько важных ограничений, а именно: данные в наборах должны быть однотипны, то есть последовательность полей и типы данных в обоих наборах должны быть одинаковы. А также, естественно, количество выводимых колонок должно совпадать в каждом наборе.

Операторы MINUS, INTERSECT

MINUS вычитает из первого набора данных второй набор данных, то есть в результате выполнения SQL-запроса с оператором MINUS на экран будут выведены лишь те строки из первого набора, которых нет во втором наборе – в подзапросе, следующем непосредственно после оператора MINUS. Вывести те номера телефонов, которые есть в MAN, но которых нет в таблице AUTO.

```
select phonenum from man MINUS select phonenum from auto
```

Оператор INTERSECT выведет только те строки, которые есть и в первом, и во втором наборе данных, то есть пересечение множеств

```
select phonenum from man INTERSECT select phonenum from auto;
```

Обновление данных и удаление данных с использованием подзапросов

Обновление строк UPDATE

Напишем запрос для обновления данных в таблице MAN, где люди проживают в городах

с населением более миллиона жителей.

Задание звучит следующим образом.

Обновите возраст людей в таблице MAN (YEAROLD +1) у тех людей, которые проживают в городах с населением (PEOPLES) более миллиона человек.

```
update man set yearold = yearold + 1 where citycode in (SELECT citycode FROM city WHERE peoples > 100000);
```

Удаление строк DELETE

Подобный синтаксис можно использовать и при удалении данных с помощью команды DELETE. Например, удалите из таблицы AUTO1 все записи, которых нет в таблице AUTO.

```
delete auto1 where auto1.regnum not in (select regnum from auto1)
```

Команды UPDATE и DELETE требуют завершения командой COMMIT для фиксации изменений в базе данных.

Нормализация – это правила, позволяющие улучшить, оптимизировать базу данных.

Нормализация ставит задачу обеспечить поддержку целостности данных.

Нормализа-

ция – процесс, обеспечивающий хранение одного элемента в одном-единственном экземпляре

в базе данных, позволяя тем самым избегать дублирования. Считается, что база данных находится во второй нормальной форме, если соблюдаются следующие условия:

- таблица уже находится в 1НФ и при этом все неключевые атрибуты зависят только от первичного ключа, то есть
- вторая нормальная форма требует, чтобы неключевые колонки таблиц зависели от первичного ключа в целом, но не от его части;
- если таблица находится в первой нормальной форме и первичный ключ у нее состоит

из одного столбца, то она автоматически находится и во второй нормальной форме. Третья нормальная форма

Третья нормальная форма – это когда таблица находится во второй нормальной форме,

каждый неключевой атрибут зависит только от первичного ключа и такие атрибуты не зависят

друг от друга.

Вставка данных из запроса

Введение

В SQL есть возможность добавлять данные в таблицу с помощью запроса SELECT. Это особенно удобно, когда нам следует осуществить вставку довольно большого количества данных.

Создание таблиц на основе запроса

Введение

Язык SQL позволяет создавать таблицы на основе запроса. При этом в таком запросе

можно использовать группировки, операторы объединения таблиц, математические расчеты.

```
CREATE TABLE TAB1 AS SELECT * FROM TAB2 WHERE 1=0
```

PIVOT – переворачиваем запрос с группировкой

Иногда необходимо транспонировать таблицу или запрос, то есть чтобы данные, которые были в строках таблицы, стали бы столбцами. Данные в ячейках таблицы становятся заголовками столбцов.

Для транспонирования таблицы с группировкой в ORACLE SQL используется специаль-

ный оператор PIVOT.

Синтаксис конструкции:

```
SELECT * FROM
(
  SELECT fieldlist
  FROM tables
  WHERE conditions
)
PIVOT
(
  aggregate_function(field)
  FOR field2
  IN ( expr1, expr2, ... expr_n) | subquery
) ORDER BY expression [ ASC | DESC ];
```

Для функции PIVOT существует обратная функция UNPIVOT – для преобразования строки запроса в столбец.

Использование итераторов

Язык SQL позволяет программировать специальные последовательности целых чисел (итераторы), которые могут использоваться в запросах.

Создание итераторов реализуется с помощью конструкции CONNECT BY.

Синтаксис

```
SELECT LEVEL AS lv FROM dual CONNECT BY LEVEL<N
```

Вывести числа от 1 до 100, кроме чисел, которые делятся на 3 нацело.

```
SELECT * FROM (
SELECT level lv FROM dual CONNECT BY LEVEL<101) WHERE MOD(lv,3)<>0
```

Иерархические запросы CONNECT BY

Достаточно часто необходимо представить данные в иерархическом виде.

Например,

есть иерархическая система подчинения подразделений на предприятии, либо состав изделий,

когда одно изделие состоит из других, а те, в свою очередь, также состоят из более мелких деталей.

Для этого используется древовидная иерархическая система, такая, где есть идентифика-

тор и есть идентификатор родителя (родительского узла) ID, PARENTED. Для работы с подоб-

ной иерархической структурой также используется конструкция SQL – CONNECT BY.

```
select level... start with нач условие обхода дерева
connect by PRIOR nocycle recurse-condition ORDER SIBLINGS By с учетом уровня
```

где LEVEL – специальная переменная, означающая уровень иерархии;

START WITH – ключевой оператор древовидной структуры:

с какого элемента необходимо начинать обход;

INITIAL-CONDITION – условное выражение для обозначения начала обхода.

Пример: создадим и заполним таблицу.

```
CREATE TABLE parentchild(id NUMBER PRIMARY KEY, name VARCHAR2(50), idparent
NUMBER);
```

Пример позволяет показать иерархию.

```
SELECT LEVEL, lpad('_', 4*LEVEL, '_') || name as nm
FROM parentchild
START WITH idparent is null
CONNECT BY PRIOR id=idparent ORDER SIBLINGS By name
```

В данном способе воспроизведения иерархии с использованием запросов SQL CONNECT BY не работает классическая сортировка ORDER BY, попытка сортировки запроса с помощью ORDER BY ломает иерархию, поэтому для сортировки подобных запросов используется специальная директива ORDER SIBLINGS BY.

Иногда при таком способе организации данных возникают так называемые циклические ссылки, это когда родительская запись ссылается на подчиненную, а подчиненная, в свою очередь, на родительскую. Обращение в запросе к таким данным приводит к бесконечному выполнению цикла и ошибке. Избежать подобной проблемы позволяет использование специальной директивы NOCYCLE. Переделаем запрос из примера с использованием данной директивы.

```
SELECT LEVEL, lpad('_', 4*LEVEL, '_') || name as nm, SYS_CONNECT_BY_PATH(name, '/') FROM
parentchild START WITH idparent is null CONNECT BY NOCYCLE PRIOR id=idparent ORDER
SIBLINGS By name
```

Условные выражения
в SQL-запросе. DECODE/CASE

В алгоритмических языках это операторы IF и CASE. В языке SQL похожая функциональность реализуется с помощью операторов DECODE, CASE.

SELECT DeCODE (выражение, значение, значение если
выражение = значение, значение если выражение!= значение)
FROM TABLE;

```
select decode(sqrt(4),2,'равно 2','не равно 2') AS EXP from dual
```

```

case
это еще один оператор для условных выражений
CASE [ выражение ]
  WHEN значение если выражение= значение1 THEN result_1
  WHEN значение если выражение= значение2 THEN result_2
  ...
  WHEN значение если выражение= значениен THEN result_n
  ELSE result (если не равно ни одному из Значений)
END

```

```

select case sqrt(16)
  when 1 then 'равно 1'
  when 2 then 'равно 2'
  when 4 then 'равно 4'
  else 'не равно 1,2,4'
  end as cse from dual;
-- равно 4

```

Если в операторе CASE нет предложения ELSE, тогда оператор CASE вернет NULL.

Временные таблицы. Когда лучше применять
Временные таблицы в SQL выполняют разные функции, например разумно применять временные таблицы, когда необходимо сохранить предварительный результат некоторого сложного расчета, подвести предварительные итоги по списку операций или подготовить данные для отчета перед отправкой на печать. Обычные таблицы называются регулярными.

ON COMMIT DELETE ROWS используется во временных таблицах, данные которой существуют в пределах одной транзакции;

- ON COMMIT PRESERVE ROWS используется во временных таблицах, данные которой существуют в пределах одной сессии.

```

CREATE GLOBAL TEMPORARY TABLE TEMP_AUTO
(
  NO NUMBER(2,0),
  MARK VARCHAR2(14)
) ON COMMIT PRESERVE ROWS;

```

Регулярные выражения в SQL

Для работы с регулярными выражениями в ORACLE SQL используются следующие операторы: REGEXP_LIKE, REGEXP_REPLACE, REGEXP_SUBSTR, REGEXP_COUNT, REGEXP_INSTR.

```
SELECT * FROM regtest WHERE REGEXP_LIKE(dt,'[0-9]{8}')
```

```
SELECT REGEXP_REPLACE(t.dt,'[0-9]{8}','mydate') rr FROM regtest t
```

REGEXP_SUBSTR выделяет из строки заданный REGEXP шаблон

```
SELECT REGEXP_SUBSTR(t.dt,'[0-9]{8}') rr,t.dt rr FROM regtest t
```

REGEXP_INSTR определяет номер первого символа вхождения REGEXP шаблона в строку.

```
SELECT REGEXP_INSTR(t.dt,'[0-9]{8}') rr,t.dt rr FROM regtest t
```

REGEXP_COUNT

REGEXP_COUNT определяет количество вхождений REGEXP шаблона в строку.

```
SELECT REGEXP_COUNT(t.dt,'[0-9]{1}') rr,t.dt rr FROM regtest t
```

Аналитический SQL. Запросы рейтингов. Накопительный итог

В ORACLE SQL существует специальный тип запросов – аналитические запросы, или запросы с аналитическими функциями. Данные функции используют в качестве одного из аргументов набор данных, который является предварительным результатом обработки основного запроса. Данные функции используются довольно часто и применяются при обработке сложной финансовой, статистической информации.

- запросы с рейтингами;
- запросы с накопительным итогом;
- запросы для вычисления результата в рамках временного окна;
- запросы для поиска следующих и предыдущих значений после текущей записи в источнике данных по некоторому признаку.


```
SELECT аналитическая функция OVER([PARTITION партицирование...]  
ORDER BY (упорядочивание выражение 2 [...]) [{ASC/DESC}] [{NULLS FIRST/NULLS LAST}]) a  
from t
```

Аналитическая функция – одна из аналитических функций, ниже по тексту есть описа-

ние данных функций.

- OVER – специальная конструкция, показывающая базе данных, что в запросе используется аналитический SQL;
- PARTITION – партицирование – это логическая модель разделения данных в запросе, некий сегмент данных, объединенных по общему признаку (обычно это одна или несколько колонок запроса);
- ORDER BY – упорядочивание – показывает, как будут отсортированы данные в рамках заданного сегмента (ASC/DESC): порядок, по возрастанию или по убыванию.

Разберем основные аналитические функции. Всего аналитических функций более 30, но наиболее часто употребляются именно следующие:

- ROW_NUMBER () – номер строки в группе;
- LAG (f, n,m): f – имя колонки, n – предыдущее значение в группе, m – значение по умолчанию;
- LEAD (f, n,m): f – имя колонки, n – последующее значение в группе, m – значение по умолчанию;
- FIRST_VALUE (f): f – имя колонки, первое значение в группе;
- LAST_VALUE (f): f – имя колонки, последнее значение в группе;
- STD_DEV (f): f – имя колонки, значение стандартного распределения в группе;
- SUM (f): f – имя колонки, накопительная сумма по группе;
- AVG (f): f – имя колонки, среднее по группе заданных групп;
- RANK (f): f – имя колонки, относительный ранг записи в группе;
- DENSE_RANK (f): f – имя поля, абсолютный ранг записи в группе.

Запросы с накопительным итогом

В данном запросе в колонке Num формируется накопительный итог по зарплате –

аналитическая функция SUM (SAL), в рамках отдела с сортировкой по зарплате
OVER
(PARTITION BY OTD ORDER BY SAL).

```
SELECT name , otd , sal , sum(sal) OVER (PARTITION BY otd ORDER BY sal) as num FROM  
personA
```

Аналитический SQL. Конструкции окна. Первая и последняя строки
В запросах с использованием аналитического SQL есть возможность работать с
окном данных и использовать записи из набора данных, которые являются
следующими либо предыдущими по отношению к текущей записи.

```
SELECT аналитическая функция OVER([PARTITION партицирование...]  
ORDER BY (упорядочивание выражение 2 [...] [{ASC/DESC}] [{NULLS FIRST/NULLS LAST}]) a  
from t  
ROWS | RANGE] [{UNBOUNDED | выражение} PRECEDING | CURRENT ROW  
ROWS | RANGE]  
BETWEEN  
UNBOUNDED PRECEDING | CURRENT ROW |  
UNBOUNDED | выражение 1}{PRECEDING | FOLLOWING  
AND  
UNBOUNDED FOLLOWING | CURRENT ROW |  
UNBOUNDED | выражение 2}{PRECEDING | FOLLOWING
```

Здесь PRECEDING и FOLLOWING задают верхнюю и нижнюю границы окна
агрегирования (то есть определяют строки интервала для агрегирования).

```
SELECT name , otd , sal  
, avg(sal) over (PARTITION BY otd order by sal ROWS BETWEEN 3 PRECEDING AND CURRENT  
ROW) as num FROM personA
```

Разберем данные аналитические функции:

- LAG (f, n, m): f – имя поля, n – предыдущее значение в группе, m – значение по умолчанию;
- LEAD (f, n, m): f – имя поля, n – последующее значение в группе, m – значение по умолчанию;
- FIRST_VALUE (f): f – имя поля, первое значение в группе;
- LAST_VALUE (f): f – имя поля, последнее значение в группе.

Несколько примеров использования данных аналитических функций.

```
SELECT name , otd , sal
, lag(sal,1) OVER (PARTITION BY otd ORDER BY sal desc) as lag
, lead(sal,1) OVER (PARTITION BY otd ORDER BY sal desc) as lead FROM personA
```

Выводится зарплата сотрудника, предыдущая зарплата по рейтингу LAG (SAL,1), следующая зарплата по рейтингу LEAD (SAL,1).

```
SELECT name , otd , sal
, FIRST_VALUE(sal) OVER (PARTITION BY otd ) as first , LAST_VALUE(sal) OVER (PARTITION BY
otd ) as last FROM personA
```

В функциях LAG и LEAD есть еще один важный параметр – это значение, которое будет возвращать функция, если результат будет равен NULL.

В данном случае пустые значения функций LAG, LEAD заменятся на 9999999999.

Обратите внимание, что при использовании FIRST_VALUE, LAST_VALUE указывать сортировку не обязательно. Тогда данные функции будут работать как агрегатные функции MIN и MAX в рамках сегмента. При указании сортировки LAST_VALUE будет возвращать последнее значение в рамках накопительного итога

Конструкция KEEP FIRST/LAST

Конструкция KEEP FIRST/LAST используется в SQL ORACLE для вычисления значения первой или последней записи в заданной подгруппе, отсортированной по некоторому признаку. Также позволяет найти результат агрегатной функции по сгруппированным данным, если таких значений несколько.

```
SELECT
ticker,
min(price) KEEP (DENSE_RANK FIRST ORDER BY pdate) as minfirstprice,
max(price) KEEP (DENSE_RANK FIRST ORDER BY pdate) as maxfirstprice,
min(price) KEEP (DENSE_RANK LAST ORDER BY pdate) as minlastprice,
max(price) KEEP (DENSE_RANK LAST ORDER BY pdate) as maxlastprice
FROM prices GROUP BY ticker ORDER BY ticker;
```

В конструкции KEEP FIRST, KEEP LAST мы можем использовать следующие агрегатные функции: MIN, MAX, SUM, AVG, COUNT, VARIANCE, STDDEV.

Конструкция WITH

Для повышения наглядности SQL и читаемости запросов SELECT, для удобства разработки начиная с версии 9 в SQL диалекта ORACLE добавлен специальный оператор WITH.

Синтаксис Простой WITH

```
With t1 псевдоним подзапроса  
as ( select ... -- внутренни подзапрос  
) select перечень полей или * from t1 -- обращение в внутреннему подзапросу
```

Сложный WITH

```
WITH  
T1 as (SELECT field_list FROM T list join WHERE cond group by ...),  
T2 as (SELECT field_list FROM T list join WHERE cond2 group by ..), tn as ....  
SELECT * FROM T1, T2 where t1.cond= t2.cond
```

Примеры

Простой оператор WITH:

```
WITH T AS (SELECT CITYCODE FROM CITY) , T2 AS  
(SELECT CITYNAME FROM CITY A , T WHERE T.CITYCODE = A.CITYCODE)  
SELECT CITYNAME FROM T2
```

Сложное обращение

```
WITH T AS (SELECT * FROM Auto WHERE mark = 'BMW') ,  
T2 AS (SELECT * FROM Auto1)  
SELECT * FROM T, T2 WHERE T.regnum = T2.regnum;
```

Два запроса из WITH

```
WITH  
t1 as (SELECT object_type, created Istcrdt , object_name FROM all_objects),  
t2 as (SELECT object_type, count(object_name) cnttype FROM t1 group by object_type)  
SELECT * FROM t1 INNER JOIN t2 on t1.object_type = t2.object_type
```

Конструкция WITH и функции

В SQL диалекте ORACLE 12C есть возможность определить функцию или процедуру на языке PL/SQL и использовать с помощью оператора WITH, как обычный SQL. Специальная инструкция WITH с FUNCTION используется, когда необходимо вернуть данные, преобразованные с помощью сложного нелинейного алгоритма.

```

WITH
PROCEDURE <NAME_PROCEDURE>
BEGIN
...
END;

FUNCTION <NAME_FUNCTION>
BEGIN
...
END;
SELECT <NAME_FUNCTION>
FROM <TABLE>;

```

Вывести на экран марки автомобиля из таблицы AUTO.

```

WITH
FUNCTION reversive_fnc(p_name VARCHAR2) RETURN VARCHAR2
is i NUMBER; v VARCHAR2(50);
begin
FOR i IN 1..LENGTH(p_name) LOOP
v := v || SUBSTR(p_name, LENGTH(p_name)-i+1, 1);
END LOOP;
return v;
end;
SELECT DISTINCT reversive_fnc(mark) as rname, mark FROM auto

```

В функциях WITH можно использовать динамический SQL и курсоры, однако нельзя использовать команды модификации данных.

В ORACLE существует системное представление ALL_OBJECTS.

Необходимо посчитать количество таблиц, индексов, представлений для каждого владельца и общее количество объектов каждого владельца из представления.

```

SELECT owner,
       count(object_name) object_name,
       count(decode(object_type, 'TABLE', 1, null)) tablecount,
       count(decode(object_type, 'INDEX', 1, null)) idxcount,
       count(decode(object_type, 'VIEW', 1, null)) viewcount
FROM all_objects WHERE object_type in ('TABLE','INDEX','VIEW') AND rownum<100

```

Преобразуем запрос в строчку LISTAGG

В ORACLE SQL есть возможность преобразовать результат запроса в CSV-строку или в любую другую заданную строку с разделителем.

Функция LISTAGG объединяет значения заданного поля таблицы для каждой группы в строку, через указанный разделитель.

```
SELECT LISTAGG (поле [, 'разделитель'])  
  WITHIN GROUP (order_by_clause сортировка) [OVER partition]
```

```
SELECT tname, LISTAGG(phone, ' ; ') WITHIN GROUP (ORDER BY phone) phonestr FROM  
cl_phones GROUP BY tname;
```

Работаем с JSON

В ORACLE существует возможность работать с JSON-форматом.

В ORACLE 12 существует специальный тип ограничений, чтобы добавлять в колонку только данные JSON-формата и исключить ошибки, связанные с нарушением структуры JSON. Добавим такую колонку к нашей таблице:

```
ALTER TABLE info_user_v ADD CONSTRAINT c_1_json_data CHECK(json_data is json);
```

Добавляем записи:

```
INSERT INTO info_user_v(id, name, json_data)  
VALUES  
(1, 'item1', '{  
  "title": "book",  
  "name": "Island",  
  "autor": {  
    "firstname": "Robert",  
    "lastname": "Stivenson"  
  }  
}');  
  
INSERT INTO info_user_v (id, name, json_data)  
VALUES  
(2, 'item2', '{  
  "title": "new book",  
  "name": "Game of trones",  
  "autor": {  
    "firstname": "Robert",  
    "lastname": "BArateon"  
  }  
}');
```

```
SELECT id, json_value(json_data,'$.title') FROM info_user_v
```

Запрос с использованием JSON_VALUE, JSON_DATA

```
SELECT id, json_value(json_data,'$.autor.firstname')  
, json_value(json_data,'$.autor.lastname') FROM info_user_v WHERE  
json_value(json_data,'$.title') like 'bo%'
```

Еще один способ:

```
SELECT id,tt. * FROM info_user_v  
, JSON_TABLE(json_data,'$.autor' COLUMNS(firstname varchar2(50 char) path '$.firstname'))  
tt;
```

Высший пилотаж SQL. MODEL

Оператор MODEL в ORACLE SQL позволяет определенным образом эмулировать работу с электронными таблицами.

Оператор SQL MODEL позволяет рассматривать результат запроса как многомерный массив. При этом в SQL задаем оси измерения этого массива (идентифицируем данные по осям). Использование MODEL также позволит нам подводить промежуточные и общие итоги с применением агрегатных функций.

```
SELECT *FROM table1 -- таблица или запрос  
MODEL DIMENSION BY (field1 , field2, ..)--оси, определение осей измерений по которым  
мы строим массив (поля для поиска уникальной ячейки)  
MEASURES (field3) -- определяющее поле  
RULES ( cnt['res1', 'res2'] = res3 -- результат который вносится массив  
 ) ORDER BY field1; -- сортировка по полю
```



```

SELECT * FROM pen
MODEL DIMENSION BY (prt, color) -- измерения оси строим по полям prt, color
MEASURES (cnt) -- работаем с cnt
RULES (
  cnt[any, 'red'] = cnt[cv(prt), 'red'] * 10 -- для каждого prt и color = red - cnt в итоговом
  запросе умножаем на 10
)
ORDER BY prt;

```

MODEL-аналитика, сложные последовательности и массивы

MODEL также можно использовать для создания двумерных массивов, а также сложных итераторов, сложных последовательностей.

Мы можем также использовать MODEL для создания массивов.

Создание одномерного массива:

```

SELECT *
FROM dual
MODEL DIMENSION BY (0 as t1)
MEASURES (cast(dummy as varchar2(20)) as ct)
RULES (
  ct[0] = '1',
  ct[1] = '2',
  ct[1] = '3',
  ct[2] = '4',
  ct[3] = '5'
) order by 1;
--T1    CT
--0      1
--1      3
--2      4
--3      5

```

TIMESTAMP и DATE

TIMESTAMP – специальный тип данных в СУБД ORACLE, расширяющий возможности типа данных DATE. TIMESTAMP используется на практике достаточно часто, поэтому данному типу данных посвящен отдельный шаг.

Данные типа TIMESTAMP. Для работы с типом TIMESTAMP существует специальная функция CURRENT_TIMESTAMP, которая возвращает текущее значение время-дата с данными часового пояса, заданного в параметрах сессии.

```

ALTER SESSION SET TIME_ZONE = '-3:0';
select CURRENT_TIMESTAMP from dual;

```


Фрагментация таблиц, секционирование

ORACLE поддерживает секционирование таблиц. Секционирование позволяет разделить большую таблицу на более маленькие части по определенному логическому принципу.

Фрагментация (секционирование) – это разделение таблицы или индекса на несколько логически связанных частей, фрагментов, секций с неким общим признаком. Допустим, у нас есть таблица начислений, мы разбиваем эту таблицу на множество секций, например по начислениям за каждый месяц.

Кому и зачем это нужно? Вопрос с секционированием таблиц тесно связан с другим важным вопросом – вопросом масштабируемости проекта.

С помощью фрагментации появляется возможность управления фрагментами (секциями) в больших таблицах, то есть часть ненужных нам данных в текущий момент можно перенести на сторонний носитель.

В ORACLE используется три типа фрагментации (партиционирования) для таблиц.

Фрагментация по диапазону значений

Фрагментация по списку значений

Фрагментация с использованием хэш-функции

Фрагментация по данным заданных столбцов таблицы. ORACLE вычисляет значение специальной хэш-функции, на основании которого определяет, в какой именно фрагмент таблицы поместить заданную запись.

Совмещенный тип фрагментации

Тип фрагментации, совмещающий в себе фрагментацию с использованием хэш-функции и фрагментацию по диапазону значений.

Синтаксис

Для создания и фрагментации таблиц используется дополнительная синтаксическая конструкция в команде CREATE TABLE – PARTITION BY.

Обычный синтаксис для создания ферментированной таблицы выглядит следующим образом:

```
CREATE TABLE Имя таблицы  
(столбец1 тип, столбец2 тип, столбец n)  
PARTITION BY HASH(имя столбца по которому строится хэшфункция)
```

С помощью оператора SELECT есть возможность как выбирать все данные из фрагментированной таблицы, так и использовать SELECT для выбора данных из заданного фрагмента таблицы.

```
select * from таблица partition(фрагмент);
```

Создадим таблицу проводок с фрагментацией по диапазону значений.

```
CREATE TABLE pro_range
( summ int,
  docdate date,
  docnum number
)
PARTITION BY RANGE(docdate)
(partition pt_1 values less than (to_date('01.02.2014','DD.MM.YYYY')) tablespace TBLSP1,
 partition pt_2 values less than (to_date('01.03.2014','DD.MM.YYYY')) tablespace TBLSP2,
 partition pt_3 values less than (to_date('01.04.2014','DD.MM.YYYY')) tablespace TBLSP3,
 partition p_othertext values less than (maxvalue) tablespace TBLSP3
);
```

Фрагментация с использованием хэш-функции

Создадим таблицу проводок с фрагментацией по хэш-функции.

```
CREATE TABLE pro_hash
( summ int,
  docdate date,
  docnum number
)
PARTITION BY HASH(docnum)
(partition pt_1 tablespace TBLSP1,
 partition pt_2 tablespace TBLSP2,
 partition pt_3 tablespace TBLSP3
);
```

Смешанный тип фрагментации

Смешанный тип фрагментации предусматривает как фрагментацию по диапазону значений, так и дополнительную фрагментацию по хэш-функции или фрагментацию по списку значений.

```

CREATE TABLE pro_range_hash
( summ int,
  docdate date,
  docnum number
)
PARTITION BY RANGE(docdate)
SUBPARTITION BY HASH(docnum)
SUBPARTITION TEMPLATE(
  SUBPARTITION spt_1 TABLESPACE TBLSP1,
  SUBPARTITION spt_2 TABLESPACE TBLSP2,
  SUBPARTITION spt_3 TABLESPACE TBLSP3
)
(partition pt_1 values less than (to_date('01.02.2014','DD.MM.YYYY')) tablespace TBLSP1,
 partition pt_2 values less than (to_date('01.03.2014','DD.MM.YYYY')) tablespace TBLSP2,
 partition pt_3 values less than (to_date('01.04.2014','DD.MM.YYYY')) tablespace TBLSP3,
 partition p_othermax values less than (maxvalue) tablespace TBLSP3
);

```

Работаем с XML в SQL

Сложные группировки SET GROUP CUBE

В ORACLE существуют конструкции для подведения подытогов и итогов групповых операций. Это операции CUBE, ROLLUP. Используются вместе с группировкой GROUP BY.

```

SELECT выражение1, выражение2, выражение3
FROM Man GROUP BY ROLLUP(выражение2)

```

ROLLUP – выражение, по которому подводятся итоги.

GROUPING – условное выражение, которое показывает, является ли строчка строчкой подведения итогов. Очень удобно использовать ROLLUP CUBE при подведении итогов.

```

SELECT substr(firstname,1,2) as Caps, count(firstname) countr FROM Man GROUP BY
substr(firstname,1,2)

```

Итоги нашей выборки подведем с помощью ROLLUP.

```

SELECT substr(firstname,1,1) Caps, count(firstname) countr FROM Man GROUP BY
ROLLUP(substr(firstname,1,1))

```

SELECT substr(firstname,1,1) Caps, count(firstname) countr FROM Man GROUP BY ROLLUP(substr(firstname,1,1))	
Results	Explain Describe Saved SQL History
..	.
M	1
S	1
A	1
M	1
P	2
T	1
-	13

Есть еще дополнительная функция GROUPING SET, поясните ее Работу. Предложение GROUPING SETS – это расширение предложения GROUP BY, которое можно использовать, чтобы задать одновременно сразу несколько группировок данных. Пример работы:

```
SELECT mark, color, count(1)
FROM auto1 GROUP BY GROUPING SETS (mark, color, (mark, color));
```

Представления

Ранее мы уже разбирали конструкцию WITH, когда можно предварительно составить запрос и обратиться через псевдоним к этому запросу. Подобный функционал обеспечивают представления VIEW.

Представления являются физическим объектом в базе данных, который создается на основе запроса. Имя представления не может начинаться с цифр и должно быть уникально в рамках схемы, а также не должно совпадать с именем любой из таблиц или зарезервированным словом СУБД. Обычное представление VIEW физически не содержит данных, запрос из представления выполняется приблизительно, так же как обычный запрос.

```
CREATE or REPLACE VIEW viewname AS SELECT * или колонки FROM таблицы WHERE
условия GROUP BY группировка если надо OR ORDER BY сортировка
```

где VIEWNAME – название представления, AS – запрос-обращение к представлению.

```
CREATE OR REPLACE VIEW v_auto  
AS SELECT a.color, a.mark as mark FROM auto a where color = 'СИНИЙ';
```

Выбор данных из созданного представления v_AUTO.

```
SELECT * FROM v_auto;
```

На чтение данных из представлений нужны права, которые может выдать администратор системы.

- На создание представлений также нужны права, которые может выдать администратор системы.
- Представления не сохраняют результата запроса, а только текст запроса, каждый раз при обращении к заданному представлению-запросу выполняются вновь.
- Для создания отчетов лучше использовать материализованные представления или временные таблицы.

Синонимы

Синонимы (synonyms) – специальные псевдонимы объектов базы данных, применяются для удобства доступа к объектам других схем базы данных, могут использоваться для распределения прав и безопасности доступа к данным.

Синонимы могут быть приватными (private) или общедоступными (public).

Общедоступные синонимы видны всем пользователям базы данных, приватные синонимы принадлежат только заданной схеме. Синонимы создаются для следующих объектов базы данных ORACLE: таблицы, представления, материализованные представления пакетов и процедур.

В каком случае это целесообразно?

- Во-первых, при обращении к объекту из разных схем для более простой формы записи.
- Для разграничения доступа, безопасности объекта.
- Во-вторых, если объект называется достаточно сложно и для того чтобы упростить обращение к этому объекту.
- Когда обращение к объекту происходит через DB LINK.

Для создания публичного синонима необходима роль CREATE PUBLIC SYNONYM;

```
CREATE PUBLIC SYNONYM синоним FOR имя_объекта;
```

Для создания публичного синонима используется директива PUBLIC перед словом SYNONYM

```
CREATE SYNONYM a9 FOR auto;
```

Обращение к данным через синоним a9:

```
SELECT * FROM a9;
```

Ретроспективные запросы

В ORACLE есть возможность узнать, какие данные были в таблицах определенное время назад. Мы можем использовать эти данные, сохранить эти данные во временные таблицы. Такие запросы называются FLASHBACK QUERY.

FLASHBACK QUERY – специальная возможность, которая позволяет видеть данные в базе таким образом, как будто это было сделано в заданный момент времени в прошлом.

Есть возможность использовать FLASHBACK QUERY двумя способами:

- специальная конструкция AS OF в SELECT-запросе;
- специальный встроенный пакет DBMS_FLASHBACK.

Конструкция:

```
select *  
  from tablename as of scn timestamp_to_scn(to_timestamp(time,'DD/MM/YYYY  
HH24:MI:SS')) ;
```

Здесь

- SELECT * FROM TABLENAME – запрос к таблице, где необходимо посмотреть данные;
- of scn TIMESTAMP_TO_SCN (TO_TIMESTAMP (time, «DD/MM/YYYY HH24:MI:SS»)) – специальная конструкция, где необходимо указать дату-время

Ретроспективный запрос

```
select *  
  from obj_t as of scn timestamp_to_scn(to_timestamp('19/04/2018 17:11:00','DD/MM/YYYY  
HH24:MI:SS')) ;
```

ORACLE DATABASE LINK и соединение с другой базой данных
DATABASE LINK (связь) – это специальный механизм для связи локальной базы данных с удаленной базой данных. Связь устанавливается только с той стороны, где создан объект DB LINK.

При создании DATABASE LINK (связь) задаются параметры учетной записи удаленной базы данных, с которой устанавливается соединение. Соответственно, и права в этой базе данных после соединения будут идентичны привилегиям учетной записи, параметры которой были заданы при создании DATABASE LINK. DATABASE LINK могут применяться для выбора записей из таблиц распределенной базы данных, а также для вставки данных в собственную локальную базу из удаленной базы. Также DB LINK применяются для вставки, обновления записей непосредственно в удаленной базе данных. Механизм ORACLE DATABASE LINK позволяет работать одновременно с множеством баз данных как с единой базой данных. Надо установить специальные аддоны и получить специальные права
Выбираем сведения об автомобилях в автосалонах в разных базах.

```
SELECT 'LOCAL', mark, color, phonenum,...  
  FROM auto - из локальной базы данных  
UNION ALL  
  SELECT 'Moscow', mark, color, phonenum  
    FROM auto@london  
UNION ALL  
  SELECT 'Kiev', mark, color, phonenum  
    FROM auto@tokyo  
UNION ALL  
  SELECT 'Voronej', mark, color, phonenum  
    FROM auto@sydney  
UNION ALL  
  SELECT 'Minsk', mark, color, phonenum  
    FROM auto@la;
```

Корзина в ORACLE

Начиная с версии ORACLE 10g, в ORACLE появился новый механизм корзины RECYCLE BIN, объекты после удаления теперь можно восстановить.

```
SELECT * FROM recyclebin;  
корзина текущей схемы  
  
SELECT * FROM dba_recyclebin;  
корзина для всех схем  
  
очистка корзины:  
-- чистим свою  
PURGE RECYCLEBIN;  
-- чистим все  
PURGE DBA_RECYCLEBIN;
```

Отключение корзины:

```
ALTER SESSION SET recyclebin=OFF;
```

ВОССТАНОВЛЕНИЕ ТАБЛИЦЫ ИЗ КОРЗИНЫ:

```
FLASHBACK TABLE my_dropped_table TO BEFORE DROP;
```

Пример

```
create table t as select owner, count(object_name) cnt from all_objects group by owner;  
drop table t;  
FLASHBACK TABLE t TO BEFORE DROP;  
select * from t;
```

Массовая операция вставки данных

Для множественной вставки данных в SQL ORACLE существует команда INSERT ALL.


```

INSERT ALL
  WHEN (<condition>) THEN
    INTO <table_name> (<column_list>)
    VALUES (<values_list>)
  WHEN (<condition>) THEN
    INTO <table_name> (<column_list>)
    VALUES (<values_list>)
  ELSE
    INTO <table_name> (<column_list>)
    VALUES (<values_list>)
  SELECT <column_list> FROM <table_name>;

```

Пример

```

INSERT ALL
  WHEN object_type = 'TABLE' THEN
    INTO obj_t values (owner, object_name)
  WHEN object_type = 'INDEX' THEN
    INTO obj_i values (owner, object_name)
  WHEN object_type = 'CLUSTER' THEN
    INTO obj_c values (owner, object_name)
  select owner, object_type, object_name from all_objects where rownum<9900;
select * from obj_t;

```

- INSERT ALL – операция модификации данных, после ее завершения необходимо выполнить COMMIT.
- INSERT ALL может использоваться для вставки данных только в таблицы, но не в представления или материализованные представления.
- Сумма столбцов во всех предложениях INSERT INTO не должна превышать 999.
- INSERT ALL нельзя использовать для таблиц с коллекциями.
- В INSERT ALL нельзя использовать последовательность SEQUENCE.

Массовое обновление данных

Существует интересная возможность обновления данных в связанных таблицах с помощью команды UPDATE. Это способ множественного обновления данных на основе запроса.

```

UPDATE
(
SELECT
    t.c1, t.c2, s.c1 AS c1_new, s.c2 AS c2_new
FROM table1 t
INNER JOIN table2 s ON s.key=t.key
) tt SET tt.c1=tt.c1_new, tt.c2=tt.c2_new

create table PHONES1
(
    PHONENUM VARCHAR2(48) not null primary key,
    NAME VARCHAR2(48)
);

create table PHONES
(
    PHONENUM VARCHAR2(48) not null primary key,
    NAME VARCHAR2(48)
);

```

Необходимо отметить, что такую команду обновления данных следует также завершать командой COMMIT.

Команда MERGE

Существует команда MERGE, которая одновременно выполняет вставку и обновление данных в одной таблице на основе данных из другой таблицы. При слиянии таблиц проверяется условие, и если оно истинно, то выполняется UPDATE, а если нет – INSERT.

```

MERGE INTO таблица USING (select запос) ON (condition)
WHEN MATCHED THEN
    UPDATE SET column2 = value1 [, column2 = value2 ...]
WHEN NOT MATCHED THEN
    INSERT (column1 [, column2 ...]) VALUES (value1 [, value2 ...]);

```

Только вставка

```

merge into t_obj mt using (select owner, object_type, count(1) as objcount from all_objects
group by owner, object_type) t
on (mt.owner = t.owner and mt.object_type = t.object_type )
WHEN NOT MATCHED THEN
INSERT (mt.owner, mt.object_type, mt.objcount)
VALUES (t.owner, t.object_type, t.objcount);

```

Только обновление

```
MERGE INTO t_obj mt using (select owner, object_type, count(1) as objcount from all_objects
group by owner, object_type) t
on (mt.owner = t.owner and mt.object_type = t.object_type )
WHEN MATCHED THEN
UPDATE SET mt.objcount = t.objcount;
```

Эту команду следует завершать командой COMMIT.

Транзакции и блокировки

Для данного шага нам потребуется установить дополнительное программное обеспечение (ORACLEXE, PL SQL DEVELOPER).

Режим SERIALIZABLE

В СУБД ORACLE есть возможность, чтобы пользователь всегда видел только те данные в таблицах, которые были с начала его сессии.

Материализованные представления

Материализованные представления – это объект базы данных, который содержит результат выполнения запроса. Отличие материализованных представлений от обычных представлений в том, что рассчитанные данные в материализованных представлениях статичны, физически располагаются в табличном пространстве и обновляются по заданному алгоритму. Материализованные представления используются при построении отчетности и позволяют ускорить построение отчетов с помощью запросов SQL в несколько десятков раз, позволяя за считанные секунды оперировать миллионами записей. Также использование материализованных представлений позволяет заранее вычислить итоги финансовых операций при построении отчетности, что многократно увеличивает Производительность. В материализованных представлениях предварительно вычисляются и сохраняются результаты запроса к базе данных, в этом запросе при желании могут быть использованы соединения, агрегирования, результаты сложных математических расчетов.

```
CREATE MATERIALIZED mv_name BUILD OPTION
REFRESH OPTION
ENABLE QUERY REWRITE
AS SELECT * FROM table(SELECT Query )
```

Рассмотрим некоторые важные конструкции оператора CREATE MATERIALIZED VIEW. BUILD OPTION может принимать следующие значения:

- BUILD IMMEDIATE позволяет сразу заполнить материализованное представление данными из запроса; значение по умолчанию;

- BUILD DEFERRED разрешает загрузить данные в материализованное представление позднее, в указанное время.

REFRESH OPTION

REFRESH FAST для фиксации всех изменений главных таблиц – необходимы журналы материализованных представлений.

COMMIT конструкции REFRESH указывает на то, что все зафиксированные изменения главных таблиц распространялись на материализованное представление немедленно после фиксации этих изменений.

ENABLE QUERY REWRITE показывает оптимизатору ORACLE переписать все запросы с использованием материализованных представлений вместо лежащих в основе представления таблиц.

AS SELECT * FROM – запрос, на основании которого строится материализованное представление. Для обновления данных в материализованном представлении может быть использована следующая команда:

```
begin
  dbms_mview.refresh(viewname);
end;
```

Создадим две таблицы, заполним данными и создадим на основе этих таблиц материализованное представление.

```
create table tabm as select level * 10 as id, 'l' || to_char(mod(level, 30)) as txt from dual
connect by level<10001 ;
create table tabp as select level as id , level * 10 as pid from dual connect by level<100001 ;
```

Создание журналов для материализованного представления.

Журнал материализованного представления необходим для корректного быстрого обновления (FAST) для материализованного представления.

В нашем случае для этого нужно создание журнала для каждой из двух таблиц.

```
CREATE MATERIALIZED VIEW LOG
ON tabm WITH SEQUENCE, ROWID
(id,txt)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON tabp
WITH SEQUENCE, ROWID
(id, pid)
INCLUDING NEW VALUES;
```

Создаем материализованное представление на основе этих двух таблиц.

```
CREATE MATERIALIZED VIEW test_mv  
BUILD IMMEDIATE  
REFRESH FAST  
ENABLE QUERY REWRITE  
AS  
SELECT tabm.id, tabm.txt as txt, count(*) as cntm, count(tabm.txt) cnt FROM tabm INNER JOIN  
tabp On tabm.id = tabp.pid GROUP BY tabm.id, tabm.txt;
```

Выполним запрос к материализованному представлению.

```
select * from test_mv
```

Если вы считаете, что материализованное представление не нужно, можете уничтожить его с помощью оператора DROP MATERIALIZED VIEW:

```
DROP MATERIALIZED VIEW sales_sum_mv;
```

ежедневная банковская отчетность: ежедневно обновляются материализованные представления, где считаются предварительные итоги, после чего с их использованием строится отчетность.

Контекст сеанса

Получаем информацию о тех, кто заходил и что делал

Планировщик JOB-заданий. Управление

В ORACLE есть механизм, позволяющий запланировать выполнение определенной программы на заданное время, такая программа может быть оператором SQL, программой на языке PLSQL либо даже внешней программой. Этот механизм называется механизмом заданий JOB. Данный механизм может применяться, например, для планирования запуска тяжелых запросов в ночное время или на выходные дни, распределения задач построения отчетности.

Таблицы в ORACLE SQL. Дополнительные сведения

Но в ORACLE СУБД есть и другие виды таблиц. Это таблицы, организованные по индексу, и кластеризованные таблицы.

Индекс-таблицы (INDEX-ORGANIZED TABLE, IOT) – такие таблицы, в которых данные хранятся в виде индексной структуры, первичный ключ представляет собой индекс на основе В-дерева.

Индекс-таблицы отличаются от традиционных тем, что в индексных таблицах данные хранятся в упорядоченном виде. Данные сортируются по первичному

ключу. Индексные таблицы имеют уникальную идентификацию по первичному ключу, не могут быть с колонками типа CLOB, BLOB. Индексные таблицы не могут быть в кластерах. Доступ к данным к индексных таблицах осуществляется быстрее.

```
CREATE TABLE iot_tablename(  
  columnname_id NUMBER,  
  colname2 VARCHAR2(30),  
  colnameN VARCHAR2(120),  
  CONSTRAINT pk_columnname_new PRIMARY KEY (columnname_id))  
  ORGANIZATION INDEX;
```

Здесь

- IOT_TABLENAME – имя создаваемой таблицы;
 - COLNAME2...COLNAMEN – наименование колонки;
 - PK_COLUMNNAME_NEW – название ключа.
- ORGANIZATION INDEX означает что таблица организована по индексу.
Пример создания таблицы, организованной по индексу:

```
CREATE TABLE auto_iot(  
  regnum NUMBER,  
  mark VARCHAR2(30),  
  color VARCHAR2(120),  
  CONSTRAINT pk_auto_iot PRIMARY KEY (regnum)) ORGANIZATION INDEX;
```

Кластер – это несколько таблиц, которые физически хранятся вместе. Обычно используется в таблицах, участвующих в запросах с объединениями JOIN. Целью кластеров таблиц является повышение производительности запросов с объединениями для этих таблиц. При создании кластеризованных таблиц необходимо создать кластер, где будут располагаться эти таблицы.

```
CREATE CLUSTER man_city(citycode NUMBER);
```

Создадим кластер для двух таблиц MANc, CITYc, являющихся копиями таблиц MAN, CITY.

Далее создаем две таблицы MANc, CITYc. Создаем две таблицы MANc, CITYc для кластера MAN_CITY:

```

CREATE TABLE MANc
(
    PHONENUM VARCHAR2(15),
    FIRSTNAME VARCHAR2(50),
    LASTNAME VARCHAR2(50),
    CITYCODE NUMBER REFERENCES CITY,
    YEAROLD NUMBER,
    PRIMARY KEY ("PHONENUM")
) CLUSTER man_city(citycode);

CREATE TABLE CITYc
(
    CITYCODE NUMBER,
    CITYNAME VARCHAR2(50),
    PEOPLES NUMBER,
    PRIMARY KEY ("CITYCODE") ) CLUSTER man_city(citycode)

```

Для ускорения производительности запросов, для экономии дискового пространства, экономии оперативной памяти используется сжатие таблиц. В сжатых таблицах выполняются запросы вставки и обновления данных, но на сжатых таблицах эти операции требуют больше ресурсов. Сжатие таблиц не рекомендуется применять в таблицах при интенсивной вставке и обновлении данных. Наиболее эффективно использовать сжатые таблицы при работе в хранилищах данных.

```

CREATE TABLE testtable(
Test1 varchar2(20)
Test2 varchar2(50))
COMPRESS FOR ALL OPERATIONS;

```

Быстрая очистка таблиц и EXECUTE IMMEDIATE

```
EXECUTE IMMEDIATE 'TRUNCATE TABLE test1'
```

Индекс (INDEX) – это специальный объект базы данных. Индекс нужен для повышения производительности поиска данных в базе. Таблицы в базе данных могут иметь большое количество строк, которые хранятся в произвольном порядке, и их поиск по заданному критерию путем последовательного просмотра таблицы строка за строкой может занимать много времени. Ускорение работы с использованием индексов достигается в первую очередь за счет того, что индекс имеет структуру, оптимизированную под поиск, – например, структуру сбалансированного дерева.

Представление базы данных – это объект базы данных, который представляет собой результат запроса к данным в таблицах базы данных. Представления заново формируют данные из запроса. Если данные в таблицах базы данных, на основе которых построен запрос представления, изменяются, данные в представлении также изменятся. Представления могут также содержать агрегированные данные. Представления могут быть построены на основе запроса к нескольким таблицам.

Триггеры являются одной из разновидностей хранимых процедур именованного кода на PL SQL. Их исполнение происходит при возникновении для таблицы какого-либо события, например вставки данных, изменения данных в таблицах. Триггеры используются для проверки целостности данных, а также для некорректных данных.

Функция – это именованная подпрограмма, которая возвращает определенное значение.

Пример создания функции:

```
CREATE OR REPLACE Function SQRTF
  ( x_in IN NUMBER )
  RETURN number
IS
  cnumber number;
BEGIN
  Cnumber := x_in*x_in
RETURN cnumber;
EXCEPTION
WHEN OTHERS THEN
  raise_application_error(-20001,'error - '||SQLCODE||' -ERROR- '||SQLERRM);
END;
```

Пример вызова функции из SQL:

```
SELECT cityname, SQRTF (peoples) FROM city;
```

Процедура – это именованная подпрограмма, которая выполняет некоторое заданное действие.

Пример создания процедуры:


```

CREATE OR REPLACE procedure SQRTF
  ( x_in IN NUMBER )
  RETURN number
IS
  cnumber number;
BEGIN
  Cnumber := x_in*x_in
  Insert into abl(c) values (cnumber);
EXCEPTION
WHEN OTHERS THEN
  raise_application_error(-20001,' error - '||SQLCODE||' -ERROR- '||SQLERRM);
END;

```

Пример вызова процедуры:

```

Begin
  Sortf(10);
End;

```

Пакет – это специальный объект базы данных, объединяющий несколько функций и процедур, состоит из тела пакета и заголовка пакета.

Пример создания пакета:

```

CREATE PACKAGE pkg_rt AS
PROCEDURE proc1 ( ename VARCHAR2);
PROCEDURE proc2 (v_id NUMBER);
END emp_actions;

CREATE PACKAGE BODY pkg_rt AS
PROCEDURE proc1 ( ename VARCHAR2)
IS
BEGIN
  Null;
END;
PROCEDURE fire_employee (v_id NUMBER) IS
BEGIN
  DELETE FROM tbl1 WHERE tid = v_id;
END fire_employee;
END pkg_rt;

```

DATABASE LINK – это специальный объект в ORACLE для соединения с другой (удаленной) базой данных, разрешающий доступ к объектам другой базы данных.

Последовательности формирование первичного ключа

Последовательность SEQUENCE в ORACLE SQL – это объект базы данных, который генерирует целые числа в соответствии с правилами, установленными во время его создания.

```
CREATE SEQUENCE seq_3
START WITH 20
INCREMENT BY -1
MAXVALUE 20
MINVALUE 0
CYCLE
CACHE 10;
```

Пример использования последовательности:

```
SELECT seq_3.next_val FROM dual;
```

- START WITH позволяет создателю последовательности указать первое генерируемое ей значение. После создания последовательность генерирует указанное в START WITH значение при первой ссылке на ее виртуальный столбец NEXTVAL
 - INCREMENT BY n определяет приращение последовательности при каждой ссылке на виртуальный столбец NEXTVAL. Если значение не указано явно, по умолчанию устанавливается 1. Для возрастающих последовательностей устанавливается положительное n, для убывающих или последовательностей с обратным отсчетом – отрицательное.
 - MINVALUE определяет минимальное значение, создаваемое последовательностью. Если оно не указано, ORACLE применяет значение по умолчанию NOMINVALUE.
 - MAXVALUE определяет максимальное значение, создаваемое последовательностью. Если оно не указано, ORACLE применяет значение по умолчанию NOMAXVALUE.
 - CYCLE позволяет последовательности повторно использовать созданные значения при достижении MAXVALUE или MINVALUE. То есть последовательность будет продолжать генерировать значения после достижения своего максимума или минимума.
- Возрастающая

последовательность после достижения своего максимума генерирует свой минимум. Убывающая последовательность после достижения своего минимума генерирует свой максимум.

Если циклический режим нежелателен или не установлен явным образом, ORACLE приме-

няет значение по умолчанию NOCYCLE. Указывать CYCLE вместе с NOMAXVALUE или

NOMINVALUE нельзя. Если нужна циклическая последовательность, необходимо указать

MAXVALUE для возрастающей последовательности или MINVALUE для убывающей.

- CACHE n указывает, сколько значений последовательности ORACLE распределяет

заранее и поддерживает в памяти для быстрого доступа. Минимальное значение этого пара-

метра равно 2. Для циклических последовательностей это значение должно быть меньше, чем

количество значений в цикле. Если кэширование нежелательно или не установлено явным

образом, ORACLE применяет значение по умолчанию – 20 значений.

```
CREATE SEQUENCE seq_1
START WITH 1
INCREMENT BY 1
NOCYCLE
CACHE 10;
```

Команда DROP SEQUENCE SEQ удаляет последовательность из базы, SEQ – имя последовательности.

Внешние таблицы – специальный механизм ORACLE СУБД, с помощью которого можно обращаться к данным, хранящимся в файлах вне базы данных, как к обычным таблицам. Для загрузки данных могут использоваться команды драйвера ORACLELoader. К ExternalTABLE не могут применяться операторы изменения данных (DELETE, INSERT, UPDATE, MERGE). Но к таким таблицам вполне могут применяться стандартные запросы SELECT с использованием групповых операций, агрегатных функций, аналитического SQL. Все это делает механизм внешних таблиц особенно эффективным для проектов DWH (хранилищ данных), при формировании ETL (процедур загрузки) для хранилищ данных.

Оптимизатор запросов, чтение плана запроса

Любой SQL-запрос выполняется по заданному плану. В плане выполнения запроса описывается последовательность операций для доступа к данным, особенности использования, последовательность операций сортировок и объединения строк таблиц. Профессиональный SQL-разработчик должен уметь читать план запроса, чтобы оценить эффективность выполнения запроса.

План запроса строит оптимизатор ORACLE, задача разработчика – анализируя план запроса, найти проблемные места в производительности и постараться решить проблемы. сервис APEX

План запроса читается с самого дальнего уровня: читается операция самого максимального уровня вложения.

На представленном плане запроса:

- Операция – операция;
- OPTION – доступ к данным;
- Объект – таблица или представление, из которого выбираются данные;
- ROWS – количество строк, которое извлекается операцией;
- TIME – условное время, за которое выполняется та или иная операция;
- COST – стоимость операций для оптимизатора.

