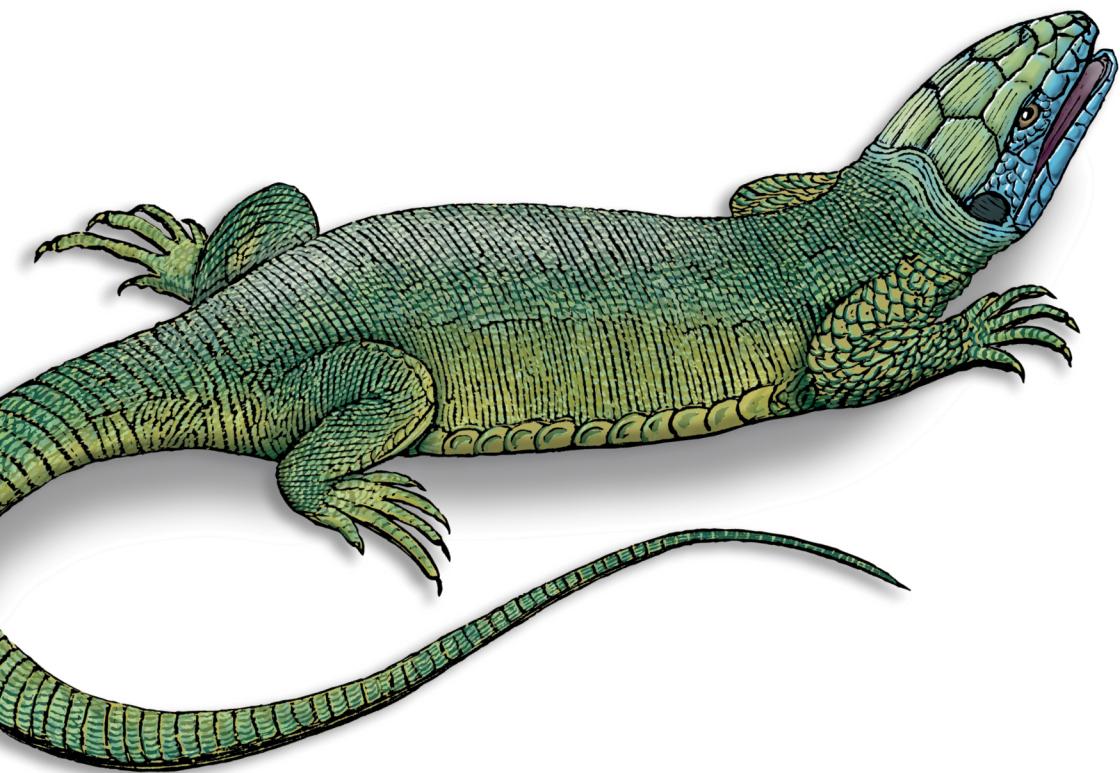


O'REILLY®

Масштабируемые данные

Лучшие шаблоны
высоконагруженных архитектур



Питхейн Стенгхольт

Data Management at Scale

Best Practices for Enterprise Architecture

Piethein Strengholt

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Масштабируемые данные

Лучшие шаблоны
высоконагруженных архитектур

Питхейн Стренгхольт



Санкт-Петербург · Москва · Минск
2022

ББК 32.973.233-018
УДК 004.62
С84

Стренхольт Питхейн

- С84 Масштабируемые данные. Лучшие шаблоны высоконагруженных архитектур. — СПб.: Питер, 2022. — 368 с.: ил. — (Серия «Бестселлеры O'Reilly»).
ISBN 978-5-4461-1461-0

Методы управления данными и их интеграции быстро развиваются, хранение данных в одном месте становится все сложнее и сложнее масштабировать. Пора разобраться с тем, как перевести сложный и тесно переплетенный ландшафт данных вашего предприятия на более гибкую архитектуру, готовую к современным задачам.

Архитекторы и аналитики данных, специалисты по соблюдению требований и управлению узнают, как работать с масштабируемой архитектурой и внедрять ее без больших предварительных затрат. Питхейн Стренхольт поделится с вами идеями, принципами, наблюдениями, передовым опытом и шаблонами.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.233-018
УДК 004.62

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1492054788 англ.

Authorized Russian translation of the English edition of Data Management at Scale ISBN 9781492054788 © 2020 Piethain Strengolt
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1461-0

© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Бестселлеры O'Reilly», 2022

Краткое содержание

Предисловие	13
Введение	15
Благодарности.....	20
От издательства	21
Глава 1. Проблемы при управлении данными.....	22
Глава 2. Введение в масштабируемую архитектуру: организация данных в масштабе	40
Глава 3. Управление большими объемами данных: архитектура хранилищ данных только для чтения.....	76
Глава 4. Управление сервисами и API: архитектура API	115
Глава 5. Управление событиями и ответами: потоковая архитектура.....	152
Глава 6. Соединение точек.....	188
Глава 7. Управление данными и их безопасность.....	223
Глава 8. Превращение данных в ценность	260
Глава 9. Управление основными корпоративными данными	293
Глава 10. Демократизация данных с помощью метаданных.....	311
Глава 11. Заключение	336
Глоссарий	346
Об авторе	366
Об обложке	367

Оглавление

Предисловие	13
Введение	15
Для кого предназначена эта книга.....	17
Что я узнаю?	18
Структура книги.....	18
Условные обозначения.....	19
Благодарности.....	20
От издательства	21
Глава 1. Проблемы при управлении данными.....	22
Управление данными	23
Аналитика фрагментирует ландшафт данных.....	26
Скорость доставки программного обеспечения меняется	28
Сети становятся быстрее	29
Приоритет — вопросы конфиденциальности и безопасности	29
Необходимо интегрировать операционные системы и системы обработки транзакций.....	30
Для монетизации данных требуется экосистемная архитектура	31
Предприятия обременены устаревшими архитектурами данных.....	32
Корпоративное хранилище данных и бизнес-аналитика.....	32
Озеро данных.....	35
Централизованное представление.....	38
Итоги главы	38
Масштабируемая архитектура	39
Глава 2. Введение в масштабируемую архитектуру: организация данных в масштабе	40
Общепризнанные отправные точки.....	41
У каждого приложения есть база данных	41
Приложения специфичны и обладают уникальным контекстом.....	41

Золотой источник.....	42
Дileммы интеграции данных не избежать	42
Приложения играют роли поставщиков и потребителей данных.....	43
Основные теоретические соображения.....	44
Принципы объектно-ориентированного программирования	44
Предметно-ориентированное проектирование	46
Бизнес-архитектура.....	49
Шаблоны связи и интеграции.....	57
Двухточечная связь	57
Разрозненные хранилища.....	58
Звездообразная модель	59
Масштабируемая архитектура	60
Золотые источники и хранилища данных предметной области	60
Контракты на поставку данных и соглашения о совместном их использовании	63
Устранение разрозненного подхода.....	64
Предметно-ориентированное проектирование в масштабе предприятия	65
Оптимизация данных для чтения.....	68
Уровень данных как целостная картина.....	69
Метаданные и целевая операционная модель	73
Итоги главы	74
Глава 3. Управление большими объемами данных:	
архитектура хранилищ данных только для чтения	76
Знакомство с архитектурой RDS.....	76
Разделение ответственности команд и запросов	77
Что такое CQRS.....	77
CQRS в масштабе	80
Службы и компоненты хранилища данных только для чтения	85
Метаданные.....	85
Качество данных	89
Уровни RDS	90
Получение данных	92
Интеграция готовых коммерческих решений.....	95
Извлечение данных из внешних API и SaaS.....	96
Служба исторических данных	97
Измерения только для добавления	100
Варианты проектирования	101

Репликация данных	103
Уровень доступа	104
Служба обработки файлов	106
Служба уведомлений о доставке	106
Служба удаления персональной информации	107
Распределенная оркестрация	107
Интеллектуальные службы потребления	108
Заполнение RDS по запросу	111
Рекомендации по использованию RDS	112
Итоги главы	113
 Глава 4. Управление сервисами и API: архитектура API	115
Знакомство с архитектурой API	115
Что такое сервис-ориентированная архитектура	116
Интеграция корпоративных приложений	120
Оркестрация сервисов	122
Хореография сервисов	125
Публичные и частные сервисы	126
Модели сервисов и канонические модели данных	127
Сходства между SOA и архитектурой корпоративного хранилища данных	128
Современный взгляд на SOA	130
API-шлюз	130
Модель ответственности	132
Новая роль ESB	134
Контракты на обслуживание	135
Обнаружение сервисов	135
Микросервисы	136
Роль API-шлюза в микросервисах	138
Функции	139
Сервисная сетка	140
Границы микросервисов	142
Микросервисы в эталонной архитектуре API	142
Коммуникация между экосистемами	143
Каналы связи на основе API	145
GraphQL	147
Метаданные	148

Использование RDS для чтения в реальном времени	
и активного чтения	149
Итоги главы	150
Глава 5. Управление событиями и ответами: потоковая архитектура.....	152
Знакомство с потоковой архитектурой	152
Асинхронная модель событий имеет значение.....	153
Как выглядят событийно-ориентированные архитектуры.....	154
Топология посредника.....	155
Топология брокера	156
Стили обработки событий	157
Введение в Apache Kafka.....	158
Распределенные события.....	161
Возможности Apache Kafka	162
Потоковая архитектура	163
Производители событий	163
Потребители событий	166
Платформа событий	168
Источники событий и команд	169
Модель управления	172
Бизнес-потоки	173
Шаблоны потребления потоковой передачи.....	176
Передача состояния с помощью событий	178
Роли RDS	179
Использование потоковой передачи для заполнения RDS.....	180
Элементы управления и политики для управления доменами	180
Потоковая передача как операционный конвейер	181
Гарантии и согласованность.....	182
Уровень согласованности	182
Семантики обработки «хотя бы один раз», «точно один раз» и «не больше одного раза».....	183
Порядок сообщений	183
Очередь недоставленных сообщений	183
Потоковое взаимодействие.....	184
Метаданные для моделей управления и самообслуживания	185
Итоги главы	186

Глава 6. Соединение точек.....	188
Кратко об архитектурах.....	188
Архитектура RDS.....	189
Архитектура API	190
Архитектура потоковой передачи	190
Усиливающие шаблоны	191
Стандарты корпоративной совместимости	193
Конечные точки устойчивых данных	193
Контракты на доставку данных	196
Доступные и адресуемые данные.....	198
Принципы пересечения сетей.....	198
Стандарты корпоративных данных.....	204
Принципы оптимизации потребления	205
Возможность обнаружения метаданных.....	208
Семантическая согласованность	212
Предоставление соответствующих метаданных.....	216
Происхождение и перемещение данных.....	217
Эталонная архитектура	219
Итоги главы	221
Глава 7. Управление данными и их безопасность.....	223
Управление данными	223
Организация: роли в управлении данными.....	225
Процессы: деятельность по управлению данными.....	228
Люди: доверительные и этические, социальные и экономические соображения	229
Технологии: золотой источник, владение приложениями и их администрирование.....	230
Данные: золотые источники, золотые наборы данных и классификации	232
Безопасность данных	239
Текущий разрозненный подход.....	240
Единая защита данных для архитектур.....	241
Поставщики удостоверений.....	243
Эталонная архитектура безопасности и подход к контексту данных	244
Процесс безопасности	246
Практическое руководство	250
Архитектура RDS	250

Архитектура API	252
Потоковая архитектура	256
Интеллектуальный механизм обучения	258
Итоги главы	259
Глава 8. Превращение данных в ценность	260
Модели потребления.....	261
Прямое использование хранилищ данных только для чтения.....	261
Хранилища данных предметной области	262
Целевая операционная модель	264
Специалисты по данным как целевая группа пользователей.....	265
Бизнес-требования.....	267
Нефункциональные требования.....	267
Построение конвейера данных и модели данных.....	269
Распространение интегрированных данных	277
Возможности бизнес-аналитики	278
Возможности самообслуживания	280
Возможности аналитики	283
Стандартная инфраструктура для автоматизированного развертывания.....	284
Модели без сохранения состояния.....	285
Предварительно настроенные рабочие места.....	285
Стандартизация шаблонов интеграции моделей.....	286
Автоматизация.....	286
Метаданные модели.....	287
Эталонная архитектура продвинутой аналитики	288
Итоги главы	292
Глава 9. Управление основными корпоративными данными	293
Демистификация управления мастер-данными	294
Стили управления основными данными.....	294
Эталонная архитектура MDM	297
Разработка решения для управления основными данными	299
Распространение MDM	300
Основные идентифицирующие номера	300
Справочные и основные данные	302
Определение области видимости корпоративных данных	302
MDM и качество данных как услуга	305

Курируемые данные.....	305
Обмен метаданными.....	306
Интегрированные представления.....	307
Повторно используемые компоненты и логика интеграции	307
Повторная публикация данных	308
Связь с управлением данными	309
Итоги главы	309
 Глава 10. Демократизация данных с помощью метаданных.....	311
Управление метаданными.....	312
Модель метаданных предприятия	313
Граф корпоративных знаний.....	321
Архитектурные подходы к управлению метаданными	325
Совместимость метаданных	326
Хранилища метаданных	328
Площадка для быстрого доступа к авторизованным данным	331
Итоги главы	334
 Глава 11. Заключение.....	336
Модель доставки	337
Полностью децентрализованный подход.....	338
Частично децентрализованный подход	339
Структурирование команд.....	339
Стратегия InnerSource.....	340
Культура	341
Выбор технологий.....	342
Упадок традиционной архитектуры предприятия	343
Чертежи и схемы.....	344
Современные навыки.....	344
Контроль и управление	344
Послесловие	345
 Глоссарий	346
Об авторе	366
Об обложке	367

Предисловие

Каждый раз, обсуждая программное обеспечение, в какой-то момент мы неизбежно переходим к разговору о данных: сколько их, где они хранятся, что означают, откуда получены или куда должны попасть и что происходит, когда они меняются. Эти вопросы оставались актуальными на протяжении многих лет, в то время как технологии управления данными быстро менялись. Современные базы данных обеспечивают мгновенный доступ к обширным наборам данных; аналитические системы отвечают на сложные наводящие вопросы; платформы потоковой обработки данных не только соединяют различные приложения, но и обеспечивают хранение информации и обработку запросов, предоставляют встроенные инструменты управления данными.

По мере развития этих технологий росли и ожидания пользователей. Пользователь часто подключается к множеству систем, расположенных в разных частях компании, когда переключается с мобильной версии на настольную, меняет местоположение или запускает одно приложение за другим. В то же время ему важна бесперебойная работа в реальном времени. Я думаю, что это значит намного больше, чем многие могут себе представить. Наша задача — связать программное обеспечение, данные и людей так, чтобы пользователи воспринимали их как единое целое.

Управление подобными системами в масштабах компании всегда было сродни черной магии, которую я освоил, помогая создавать инфраструктуру, поддержки LinkedIn. Данные в LinkedIn генерируются постоянно, 24 часа в сутки, процессами, которые никогда не останавливаются. Но когда я впервые пришел в компанию, инфраструктура обработки этих данных ограничивалась лишь медленными процедурами перебора больших дампов данных в конце дня и упрощенной поддержкой поисковых запросов, скомпонованной с доморощенными потоками данных. Концепция «пакетной обработки в конце дня» казалась мне наследием ушедшей эпохи перфокарт и майнфреймов. На самом же деле в глобальном бизнесе рабочий день не заканчивается никогда.

LinkedIn рос и становился обширным программным комплексом, и мне было ясно, что готового решения для такого рода задач не существует. Более того, управляя базами данных NoSQL, на которых работает сайт LinkedIn, я понял, что придется возродить методы распределенных систем — а это значит, что

нужно создавать решения, которых раньше не существовало. Это и привело к появлению Apache Kafka — технологии, сочетающей в себе масштабируемый обмен сообщениями, хранение и обработку обновлений профилей, посещений страниц, платежей и других потоков событий, лежащих в основе LinkedIn.

Создание Kafka не только упростило управление потоками данных в LinkedIn, но и повлияло на подходы к разработке приложений. Как и многие компании Кремниевой долины, на рубеже последнего десятилетия мы экспериментировали с микросервисами, и придумать что-то одновременно функциональное и стабильное удалось далеко не с первой попытки. Трудность заключалась в больших объемах данных, людях и программном обеспечении — сложной взаимосвязанной системе, которая должна была развиваться по мере роста компании. Чтобы справиться с такой серьезной задачей, требовались новые технологии и навыки.

Конечно, в то время не существовало инструкции по решению такой задачи. Нам приходилось изобретать все на ходу, но эта книга вполне могла бы стать тем самым руководством, в котором мы так нуждались. В ней Питхейн предлагает комплексную стратегию управления данными не просто в отдельной базе данных или приложении, а во многих БД, приложениях, микросервисах, уровнях хранения и в разных типах программного обеспечения, из которых складывается сегодняшний технологический ландшафт.

Питхейн также придерживается определенных взглядов на архитектуру, которые основаны на хорошо продуманном наборе принципов. Эти принципы позволяют отделить пространство принятия решений с помощью логических границ, внутри которых должно уместиться множество практических решений. Я думаю, что этот подход будет очень ценным для архитекторов и инженеров, поскольку в своей предметной области они сталкиваются с проблемами, описанными в этой книге. Фактически Питхейн приглашает вас в путешествие далеко за рамки данных и приложений и охватывает сложную материю взаимодействий, объединяющих целые компании.

*Джей Крепс (Jay Kreps),
соучредитель и генеральный директор Confluent*

Введение

Социальные сети, потоковая передача данных и мобильные технологии — это лишь небольшая часть нововведений, благодаря которым цифровизация за последние годы изменила нашу жизнь. Цифровизации уже подверглись музыка и телевидение, шопинг и путешествия, а также многие другие отрасли. В то же время благодаря достижениям в области искусственного интеллекта и машинного обучения развиваются автономные машины — беспилотные автомобили и летательные аппараты.

Что движет этим цифровым обществом? *Данные*. В XX веке самым ценным ресурсом была нефть. Сегодня данные — это новая нефть (<https://oreil.ly/ElYot>). Растущий спрос на аналитику поднимет спрос на данные до прежде невиданного уровня: это лишь вопрос времени.

Вместе с количеством генерируемых данных растет и их сложность. Облачные технологии, управление API, микросервисы, открытые данные, ПО как услуга (software-as-a-service, SaaS) и новые модели доставки программного обеспечения сегодня актуальны как никогда. Только за последние несколько лет было выпущено бесчисленное множество новых баз данных и аналитических приложений.

Огромное количество новых подходов к обработке данных наполняет цифровой ландшафт. Мы наблюдаем появление двухточечных интерфейсов, становимся свидетелями бесконечных дискуссий о качестве и принадлежности данных, а также множества этических и юридических дилемм, касающихся конфиденциальности, безопасности и защищенности. Гибкость, долгосрочная стабильность и четкое управление данными соревнуются с необходимостью быстрой разработки новых бизнес-задач. Эта отрасль остро нуждается в четком видении будущего управления данными и их интеграции.

Ключевая идея этой книги, касающаяся управления данными и их интеграции, основана на моем личном опыте в качестве главного архитектора данных на крупном предприятии. Благодаря этой работе я отчетливо понял, насколько хорошая стратегия в отношении данных способна повлиять на большую компанию. До этого я работал стратегическим консультантом, проектировал множество архитектур и участвовал в крупных программах управления данными, а также применял все это на практике в качестве внештатного разработчика приложений. Если вкратце, то я провел последнее десятилетие в поисках идеального решения,

которое поможет предприятиям управлять данными. Моя текущая компания, ABN AMRO Bank¹, создает то, что мы называем *перспективной архитектурой*². Мы применяем на практике и в производстве идеи, изложенные в этой книге, и учимся на этом опыте. Я знаю по книгам и по опыту, что работает, а что — нет.

Этот опыт позволяет мне описать подход к управлению данными и интеграции, выходящий далеко за рамки традиционного. Здесь вы найдете новые соединяющиеся и сочетающиеся методологии и тенденции, в том числе обзор архитектуры предприятия, бизнес-архитектуры, архитектуры программного обеспечения, предметно-ориентированного проектирования, интеграции приложений, микросервисов и облачных технологий. Эта книга представляет собой исчерпывающее руководство по созданию современной масштабируемой среды данных. Она наполнена множеством схем, принципов, шаблонов стандартизации, наблюдений, примеров и передовых практик. Вы узнаете, как не запутаться в сложном и тесно связанном ландшафте данных и встроить гибкость и контроль в ДНК вашей организации. Книга рассматривает управление данными и интеграцию с разных максимально актуальных точек зрения. В зависимости от уровня вашей организации вы можете выбрать подход, который сработает конкретно в вашем случае.

Бесчисленное множество компаний совершают ошибки в управлении данными — и это неудивительно, учитывая меняющийся ландшафт данных, их быстро увеличивающийся объем и сопутствующие проблемы интеграции. Я буду напоминать вам об этом на протяжении всей книги, уделяя пристальное внимание распространенным трудностям.

Важно отметить, что мое представление о данных можно реализовать множеством различных способов. Я упоминаю продукты и поставщиков, но общая идея не зависит от технологий. Некоторые концепции особенно сложны, и поэтому их трудно развивать. Поскольку многие области управления данными и аспекты их интеграции сильно взаимосвязаны, я буду постепенно расширять основную точку зрения в книге, начиная с базовых дисциплин, которые определяют управление данными, анализируя общую архитектуру и детально исследуя разные области.

В моем понимании архитектура должна быть долговечной, современной, распределенной, учитывающей особенности предметной области и дающей бизнесу достаточно гибкости, чтобы быстро находить и интегрировать данные, сохраняя контроль над ними. Такую архитектуру я называю *масштабируемой*.

¹ В этой книге я выражаю личные взгляды. Они не отражают точку зрения ABN AMRO.

² Перспективная архитектура близка к тому, что в OpenGroup (<https://www.opengroup.org/>) называют стратегической и целевой архитектурой. Стратегическая архитектура задает долгосрочное видение и направление для отдела развития архитектуры предприятия. Целевая архитектура — это перспектива, направление и способы развития архитектуры.

Масштабируемая архитектура отличается от других архитектур тем, что ее можно создать прагматически. Ее части можно проектировать независимо и поэтапно без больших предварительных вложений. Моя рекомендация: начинать с малого, смотреть, как идут дела, и продолжать. Этот подход разительно отличается от многих неудачных реализаций хранилищ данных, на получение хоть какой-то пользы от которых могут уйти годы.

МАСШТАБИРУЕМАЯ АРХИТЕКТУРА — ЭТО СЕТКА ДАННЫХ ИЛИ ФАБРИКА ДАННЫХ?

Идею *сеток данных* (data mesh) представила в своей статье (<https://oreil.ly/YFj8a>) разработчик и автор Жамак Дехгани (Zhamak Dehghani). Архитектура, которую описывает Дехгани, основана на современной распределенной архитектуре: предметные области рассматриваются в ней как первоочередная задача, для создания самообслуживающейся инфраструктуры данных нужно платформенное мышление, а данные интерпретируются как продукт. В некоторых отношениях она похожа на *фабрику данных* (data fabric) (<https://oreil.ly/MI45I>) — архитектуру и набор служб данных, которые обеспечивают согласованные возможности для выбора конечных точек, охватывающих локальное и облачные окружения. Наряду с этим существуют также *сервисные сетки* (service mesh) и *сетки событий* (event mesh).

С идеями и принципами, лежащими в основе этих концепций, все в порядке. Их цели совпадают. Если вам интересны эти идеи, то книга позволит глубоко погрузиться в такие дисциплины, как управление данными, безопасность данных, качество данных, управление основными данными и метаданными — в интернете такого не найти.

Для кого предназначена эта книга

Книга ориентирована на крупные предприятия, хотя может пригодиться и в небольших организациях. Ею могут особенно заинтересоваться:

- руководители и архитекторы: директора по обработке и анализу данных, технические директора, архитекторы предприятия и ведущие архитекторы данных;
- группы контроля и управления: руководители службы информационной безопасности, специалисты по защите данных, аналитики информационной безопасности, руководители по соблюдению нормативных требований, операторы баз данных и бизнес-аналитики;
- аналитические группы: специалисты по теории и методам анализа данных, аналитики и руководители аналитических отделов;
- команды разработчиков: data-инженеры, бизнес-аналитики, разработчики и проектировщики моделей данных, а также другие специалисты по данным.

Что я узнаю?

Прочитав эту книгу, вы узнаете:

- что такое управление данными и почему оно важно;
- какие тенденции в бизнесе и технологиях влияют на ландшафт данных;
- каковы основные области управления данными существуют и как они взаимосвязаны;
- как управлять сложным ландшафтом данных в масштабе;
- чем сложна интеграция данных;
- почему корпоративное хранилище данных больше не соответствует своему назначению;
- что нужно для построения масштабной архитектуры данных;
- как интерпретировать основные шаблоны распределения данных — их характеристики и некоторые варианты использования;
- какова главная роль метаданных в управлении архитектурой;
- как масштабно управлять основными и справочными данными;
- как сделать потребление данных масштабируемым с помощью шаблонов самообслуживания;
- какое влияние на архитектуру оказывают гибридное облако и пересекающиеся сети;
- как определить, какие шаблоны лучше подходят в той или иной ситуации и как их применять на практике.

Структура книги

В главе 1 объясняется, что такое управление данными и как оно развивается. В ней описывается состояние отрасли на начало 2020 года и рассказывается о взлетах и падениях основных корпоративных платформ данных.

В главе 2 мы подробно рассмотрим идею масштабируемой архитектуры. Вы познакомитесь с ее конструкцией и получите теоретические основы, на которых построена модель. В следующих главах обсуждаются особенности архитектур интеграции, составляющих общую архитектуру данных. В главе 3 рассматривается архитектура хранилищ данных только для чтения, в главе 4 — архитектура API, а в главе 5 — архитектура потоковой передачи. Глава 6 объединяет информацию из предыдущих глав и дает общий обзор.

Далее мы подробно рассмотрим, как архитектура использует более сложные аспекты управления данными и его дисциплины. В главе 7 изучаются управле-

ние данными и обеспечение безопасности с помощью способов, практических и устойчивых в долгосрочной перспективе — даже в быстро меняющихся обстоятельствах. В главе 8 рассматривается бизнес-модель масштабируемой архитектуры и показывается, как она помогает повысить ценность данных для предприятия. Глава 9 предлагает руководство по управлению основными данными с целью обеспечения их целостности в распределенных, широко-масштабных наборах, а в главе 10 подробно рассказывается об использовании, значении и потенциале демократизации метаданных. Глава 11 завершает книгу примером архитектуры предприятия и обзором будущего технологий управления данными.

Условные обозначения

В книге используются следующие типографские условные обозначения.

Курсив

Обозначает новые или важные термины.

Интерфейс

Используется для обозначения URL, адресов электронной почты.

Моноширинный шрифт

Используется для оформления листингов и фрагментов кода в тексте — имен переменных или функций, баз данных, типов данных, переменных окружения, операторов и ключевых слов, имен и расширений файлов.



Этот элемент обозначает совет или предложение.



Этот элемент обозначает общее примечание.



Этот элемент указывает на предупреждение или предостережение.

Благодарности

Я хотел бы поблагодарить Джессику Стрэнхолт-Гайтенбек (Jessica Strengholt-Geitenbeek) за то, что она позволила мне написать эту книгу. Она поддерживала меня на всем пути, заботясь о наших детях и создавая благоприятные условия для моей работы. Она — любовь всей моей жизни.

Я также хотел бы поблагодарить Сантоша Пиллаи (Santhosh Pillai), главного специалиста по архитектуре и управлению данными в ABN AMRO Bank, за доверие и за то, что он был моим наставником на протяжении всей моей карьеры. Многие идеи зародились именно в его голове. Эта книга не появилась бы без наших с ним дискуссий. Кроме того, множество других людей поддержали книгу и написали к ней отзывы: в частности, Бас ван Гилс (Bas van Gils), Дэнни Грефхорст (Danny Greefhorst), Габриэле Росси (Gabriele Rossi), Нур Спанджаард (Noor Spanjaard), Бас ван Хулсенбек (Bas van Hulsenbeek), Яцек Оффирски (Jacek Offierski), Робберт Нэстепад (Robbert Naastepad), Нил Бакстер (Neil Baxter) и др.

Наконец, спасибо замечательной команде O'Reilly за их поддержку и доверие. Сара Грей (Sarah Grey), работать с вами одно удовольствие. Ваша позитивная энергия, острый взгляд и восхитительная улыбка придавали мне сил в работе над книгой. Ким Сандовал (Kim Sandoval), спасибо за ваше умение увидеть картину в целом. Кэтрин Тозер (Katherine Tozer), большое спасибо за то, что вы довели эту книгу до конца. Мишель Смит (Michelle Smith) и Мелисса Поттер (Melissa Potter), спасибо за вашу поддержку в период адаптации.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

Проблемы при управлении данными

Управление данными усложняется из-за датафикации (<https://oreil.ly/7k6HT>). Существующие архитектуры больше не могут масштабироваться. Предприятиям нужна новая стратегия обработки данных. Без смены парадигмы и изменения культуры не обойтись, потому что централизованные решения, работающие сегодня, перестанут работать в будущем.

Технологические тенденции фрагментируют ландшафт данных. Скорость доставки программного обеспечения растет с появлением новых методологий за счет увеличения сложности данных. Быстрый рост объема данных и их интенсивное потребление перегружают операционные системы. Наконец, существуют проблемы конфиденциальности, безопасности и регулирования.

Влияние этих тенденций велико: вся отрасль должна переосмыслить подход к управлению данными в будущем. В этой книге я поделюсь другой теорией управления данными. Она идет вразрез с прежней, позволившей многим предприятиям спроектировать и организовать свой нынешний ландшафт данных. Прежде чем мы подойдем к ней в главе 2, нам нужно определить, что такое управление данными и почему оно важно. После этого мы рассмотрим общую картину и сравним различные тенденции. Наконец, мы увидим, как проектируются и организуются современные корпоративные архитектуры и платформы данных.

Для начала позвольте мне раскрыть карты. Я твердо убежден в том, что следует делать в рамках централизованного управления данными, а что — на местном уровне. Распределенная природа перспективных архитектур вдохновила меня на осмысление нового видения. Хотя хранилища и озера данных — это отличные варианты для использования информации, при их разработке не учитывались высокие темпы роста требований к потреблению данных в будущем. Например, машинное обучение потребляет значительные объемы данных, а если нужны отзывчивость и короткое время реакции, то архитектура должна быстро реагировать.

Перед тем как мы продолжим, я хотел бы попросить вас сделать глубокий вдох и отбросить предубеждения. Да, нужно стремиться к гармонизации данных, учитывать контекст при выборе их объема. При этом не стоит забывать о *масштабе*. Если экосистема сильно распределена, то централизация данных, потребляемых пользователями и приложениями, может оказаться не лучшим решением.

Управление данными

Процедуры и процессы для организации данных называются *управлением данными*. Руководство DAMA International по комплексу знаний об управлении данными (DAMA-DMBOK) (<https://oreil.ly/BKIm6>) дает более развернутое определение: «Управление данными — это разработка, выполнение и контроль планов, политик, программ и практик, которые доставляют, контролируют, защищают и повышают ценность данных и информационных активов на протяжении их жизненного цикла»¹. Очень важно глубоко внедрить эти дисциплины в вашу организацию. В противном случае вы запутаетесь и станете работать неэффективно, а ваши данные выйдут из-под контроля. В результате получение максимальной отдачи от ваших данных станет невозможным. Аналитика, например, ничего не стоит, если ваши данные некачественные.

Виды деятельности и дисциплины управления данными разнообразны и охватывают несколько областей. Некоторые из них тесно связаны с *архитектурой программного обеспечения*²: проектирование и высокоуровневая структура ПО, необходимого для разработки системы, которая отвечает требованиям бизнеса и имеет такие характеристики, как гибкость, масштабируемость, реализуемость, возможность повторного использования и безопасность. Я выбрал только аспекты управления данными, которые наиболее актуальны для современной архитектуры данных в масштабе. Поэтому здесь важно выделить следующие области.

- *Архитектура данных*. Это мастер-план данных. Она позволяет шире взглянуть на вашу архитектуру, включая схемы, эталонные архитектуры, видение перспективы и зависимости. Управление этими вещами помогает организациям принимать решения. Тематика всей книги вращается вокруг архитектуры данных в целом, но сама дисциплина и ее задачи полностью раскрываются в главах 2 и 3.

¹ Комплекс знаний разработан сообществом DAMA. Он редко обновляется: первый релиз был в 2008 году, второй — в 2014 году.

² Если вы хотите узнать больше об архитектуре программного обеспечения, я рекомендую прочитать книгу *Fundamentals of Software Architecture* Марка Ричардса (Mark Richards) и Нила Форда (Neal Ford) (O'Reilly, 2020). (Книга выходит в издательстве «Питер» в 2022 году под названием «Фундаментальный подход к программной архитектуре: паттерны, компоненты, проверенные методы». — Примеч. ред.)

- *Руководство данными.* Действия по *руководству данными* включают реализацию и обеспечение полномочий, а также контроль над управлением данными, в том числе все соответствующие активы. Эта область более подробно описана в главе 7.
- *Моделирование и проектирование данных.* Это структурирование и представление данных в определенном контексте и конкретных системах. Обнаружение, проектирование и анализ требований к данным — часть этой дисциплины. Некоторые из этих аспектов рассматриваются в главе 6.
- *Управление базами данных, хранение данных и операции с ними* относятся к управлению проектом базы данных, правильной реализации и поддержке с целью увеличения ценности данных. Эта дисциплина также включает управление операциями с базами данных. Некоторые аспекты рассматриваются в главе 8.
- *Управление безопасностью данных* включает все дисциплины и действия, обеспечивающие безопасную аутентификацию, авторизацию и доступ к данным. К таким действиям относятся предотвращение, аудит и меры по смягчению последствий распространения данных. Эта область более подробно описана в главе 7.
- *Интеграция и согласованность по данным* включают в себя все дисциплины и действия по транспортировке, сбору, объединению и трансформации данных для эффективного их перемещения из одного контекста в другой. *Согласованность по данным* — это возможность взаимодействий путем вызова функций или передачи данных между различными приложениями способами, не требующими знания характеристик приложения. С другой стороны, *интеграция данных* — это объединение данных из разных (нескольких) источников в единое представление. Этот процесс часто поддерживается дополнительными инструментами, такими как инструменты репликации и ETL (extract transform and load — «извлечение, преобразование и загрузка»), которые я считаю наиболее важными. Процесс подробно описан в главах 3, 4 и 5.
- *Управление справочными и основными данными* проводится для обеспечения доступности, точности, безопасности, прозрачности и надежности данных. Эта область более подробно описана в главе 9.
- *Управление жизненным циклом данных* относится к процессу работы с данными в течение всего жизненного цикла — от создания и первоначального хранения данных до момента их устаревания и удаления. Такой вид управления помогает эффективнее использовать ресурсы, соответствовать правовым обязательствам и ожиданиям клиентов. Некоторые дисциплины из этой области описаны в главе 3.

- *Управление метаданными.* Это управление всей информацией, которая классифицирует и описывает данные. Метаданные помогают сделать данные понятными, безопасными и готовыми к интеграции. Они также могут использоваться для обеспечения качества. Эта тема более подробно раскрыта в главе 10.
- *Управление качеством данных* необходимо, чтобы гарантировать пригодность данных к использованию. Некоторые дисциплины из этой области описаны в главах 2 и 3.
- *Управление хранением данных, бизнес-аналитикой и продвинутой аналитикой* помогает понять, как действуют бизнес-законы, и принимать решения. Эта область описана в главе 8.

Есть еще одна область DAMA-DMBOK – *интеграция и согласованность по данным* (она-то и вдохновила меня на написание этой книги). На мой взгляд, эта область недостаточно хорошо увязана с управлением метаданными, так что над ней еще предстоит поработать. Метаданные разбросаны среди множества инструментов, приложений, платформ и окружений. Они разнообразны. Согласованность метаданных – способность двух и более систем или компонентов обмениваться описательной информацией о данных – недооценена, поскольку создание крупномасштабной архитектуры и управление ею во многом связано с интеграцией метаданных. Область архитектуры данных в контексте метаданных тоже мало изучена. При правильном использовании метаданных вы четко видите, какие данные передаются, как их можно интегрировать, распределить и защитить, а также как они связаны с приложениями, бизнес-возможностями и т. д. По этой теме не хватает документации.

Я не согласен со взглядами DAMA и многих организаций на семантическую согласованность. На сегодняшний день попытки унифицировать всю семантику для обеспечения согласованности в масштабах всего предприятия продолжаются. Это называется «единственная версия истины». Однако *приложения, как и данные, всегда уникальны*. Разработка приложений часто подразумевает косвенное мышление. Вы ограничены контекстом, в котором находитесь. Этот контекст наследуется проектом приложения и находит свое отражение в данных. Мы встречаемся с ним, когда переходим от концептуальной модели проекта приложения к его логической и физической формам¹. Это важно, ведь контекст ставит в рамки будущую архитектуру. При перемещении данных между приложениями их всегда нужно преобразовывать. Даже если все данные унифицированы и хранятся централизованно, без переключения контекста не обойтись. Выхода из этой дилеммы трансформации данных не существует! Я еще вернусь к этой мысли в следующей главе.

¹ Концептуальную модель иногда также называют моделью предметной области, объектной моделью предметной области или объектной моделью анализа.

Есть еще одна точка зрения — будто управление данными должно быть централизовано и связано со стратегическими целями предприятия. Многие организации считают, что за счет этого можно снизить эксплуатационные расходы. Также существует мнение, будто централизованная платформа может избавить клиентов от проблем, связанных с интеграцией данных. Компании вложили значительные средства в корпоративные платформы данных: хранилища, озера и сервисные шины. Управление основными данными тесно связано с этими платформами, ведь объединение позволяет одновременно повышать точность наиболее важных данных.

Централизованная платформа и прилагаемая к ней централизованная модель могут потерпеть неудачу из-за развития революционных тенденций, таких как аналитика, облачные вычисления, новые методологии разработки ПО, принятие решений в реальном времени и монетизация данных. Все это ни для кого не секрет, но многие компании не осознают, какое влияние они оказывают на управление данными. Давайте рассмотрим наиболее важные тенденции и определим их масштабы.

Аналитика фрагментирует ландшафт данных

Самая заметная тенденция — это *продвинутая аналитика*. Она использует данные, чтобы сделать компании более гибкими, конкурентоспособными и инновационными. Но как продвинутая аналитика может менять существующий ландшафт данных? Чем больше данных, тем выше число вариантов и возможностей. Продвинутая аналитика — это анализ возможных вариантов, прогнозирование будущих тенденций, результатов или событий, автоматизация принятия решений, обнаружение скрытых отношений и поведения. Благодаря признанной ценности и стратегическим преимуществам продвинутой аналитики появилось множество методологий, фреймворков и инструментов для ее использования в разных направлениях. Мы только начали открывать возможности искусственного интеллекта, машинного обучения и обработки естественного языка.

Тренд, ускоривший продвинутую аналитику, — открытый исходный код. Высококачественные проекты ПО с открытым исходным кодом становятся основным стандартом¹. Открытый исходный код способствовал росту популярности продвинутой аналитики, устранив дорогостоящий аспект лицензирования коммерческих продуктов и позволив специалистам учиться друг у друга.

¹ Microsoft приобрела GitHub, популярный сервис репозиториев исходного кода, которым пользуются многие разработчики и крупные компании. IBM также признала ценность открытого исходного кода, купив RedHat, ведущего поставщика решений с открытым исходным кодом.

Открытый исходный код также открыл сферу специализированных баз данных. Cassandra, HBase, MongoDB, Hive и Redis — вот лишь часть систем, которые изменили традиционный рынок БД, позволив хранить и анализировать огромные объемы данных. В результате появления всех этих новых возможностей резко повысилась эффективность создания и разработки современных решений. Теперь сложные задачи легко можно решить с помощью узкоспециализированной базы данных. Больше не нужно использовать устаревшую реляционную БД и сложную прикладную логику. Многие из новых продуктов баз данных имеют открытый исходный код, что повысило их популярность.

Разнообразие и рост продвинутой аналитики и баз данных привели к двум проблемам: быстрому разрастанию и увеличению объема данных.

По мере разрастания одни и те же данные распределяются по множеству приложений и БД. Корпоративное хранилище, использующее систему управления реляционными базами данных (relational database management, RDBM), например, не способно выполнять сложный анализ социальных сетей. Для этой цели лучше использовать специализированную графовую базу данных¹.

Использование реляционной базы данных для централизованной платформы и связанные с этим ограничения вынуждают постоянно экспортировать данные. Таким образом, данные покидают централизованную платформу и должны быть распределены в другие системы БД. Такая ситуация может создать еще одну проблему: если данные разбросаны по всей организации, то найти и оценить их происхождение и качество труднее. Это значительно усложняет управление данными, ведь они могут распределяться и вне централизованной платформы.

Развитие аналитических методов означает ускоренный рост объема данных: соотношение чтения и записи значительно меняется. Например, аналитические модели, которые постоянно переобучаются, читают большие объемы данных. Этот аспект влияет на приложения и структуру БД, потому что приходится оптимизировать данные для чтения. Что может означать следующее: чтобы избавить системы от необходимости постоянного обслуживания данных, эти данные нужно дублировать. Это также может означать, что нужно дублировать данные для их предварительной обработки из-за разнообразия и большого количества вариантов использования и шаблонов чтения, которые сопутствуют этим вариантам. Сложно привести в порядок такое разнообразие шаблонов чтения при одновременном дублировании данных и сохранении контроля. Решение этой проблемы описано в главе 3.

¹ Этот пример использования Neo4j (<https://oreil.ly/AvEMU>) показывает, что графовая база данных — лучший вариант для проведения анализа социальных сетей.

Скорость доставки программного обеспечения меняется

В современном мире программные сервисы составляют основу бизнеса. Это означает, что нужно быстро выпускать обновления. В ответ на требования большей гибкости в таких компаниях, как Amazon, Netflix, Facebook, Google и Uber, появились новые идеологии: они развиваются свою практику разработки программного обеспечения на основе двух убеждений.

Первое убеждение: разработку программного обеспечения (*development*, *Dev*) и практическое применение информационных технологий (*operations*, *Ops*) нужно объединять. Это поможет сократить жизненный цикл разработки и обеспечить непрерывную поставку качественного ПО. Такая методология называется *DevOps*. Она требует новой культуры, предполагающей большую автономию, открытое общение, доверие, прозрачность и согласованную работу команды.

Второе убеждение касается размера разрабатываемого приложения. Ожидается, что гибкость системы возрастет, если приложения превратятся в мелкие сервисы, решающие узкоспециализированные задачи. Этот подход к разработке породил несколько модных словечек: *микросервисы* (<https://microservices.io/>), *контейнеры*, *Kubernetes* (<https://kubernetes.io/>), *предметно-ориентированное проектирование*, *бессерверные вычисления* и т. д. Я не буду вдаваться в подробности каждой концепции, но эта эволюция разработки программного обеспечения связана с увеличением сложности и спроса на улучшенный контроль данных.

Преобразование монолитного приложения в распределенное создает множество проблем, связанных с управлением данными. При дроблении приложений на более мелкие части данные распределяются по разным компонентам. Командам разработчиков приходится переводить свои (отдельные) уникальные хранилища с простыми и понятными моделями данных, на архитектуры, где объекты данных распределены. Это создает несколько сложностей: увеличение количества сетевых взаимодействий, необходимость синхронизации копий данных для чтения, обеспечение согласованности и ссылочной целостности и т. д.

Сдвиг в разработке ПО нуждается в архитектуре, позволяющей более мелким приложениям распределять свои данные. К тому же нужна новая культура *DataOps* (<https://oreil.ly/G8Bt4>) и другая философия проектирования с большим упором на совместимость данных, фиксацию неизменяемых событий, а также воспроизводимость и слабую связанность. Мы обсудим это подробнее в главе 2.

Сети становятся быстрее

Сети становятся быстрее, а пропускная способность увеличивается из года в год. Я был на саммите Gartner Data and Analytics в 2018 году, где представители Google продемонстрировали возможность перемещения сотен терабайт данных в облако менее чем за минуту.

Возможность перемещения таких больших объемов данных позволяет реализовать интересный подход: вместо увеличения вычислительной мощности источников данных, как это было принято раньше из-за сетевых ограничений, можно доставлять необработанные данные потребителям, обладающим большой вычислительной мощностью. Сеть больше не является препятствием, поэтому мы можем быстро перемещать терабайты данных из окружения в приложения, потребляющие и использующие эти данные. Интерес к такой модели неуклонно растет с увеличением популярности рынков SaaS (Software as a service — программное обеспечение как услуга) и MLaaS (Machine Learning as a Service — машинное обучение как услуга). Вместо выполнения сложных вычислений на месте можно использовать сети и передавать данные другим сторонам.

Этот шаблон распределения, основанный на копировании (дублировании) и передаче данных потребителю для обработки, находящемуся, например, в облаке, еще больше фрагментирует ландшафт данных, увеличивает важность использования четкой стратегии управления.

Приоритет — вопросы конфиденциальности и безопасности

Данные, несомненно, имеют ключевое значение для оптимизации, внедрения инноваций или дифференциации организаций. Но они также начали раскрывать себя с темной стороны, влекущей за собой негативные последствия. Файлы Cambridge Analytica (<https://oreil.ly/lLZSG>) и 500 миллионов взломанных учетных записей в Marriott — впечатляющие примеры скандалов, связанных с конфиденциальностью данных и их утечкой¹. Правительства принимают все более активное участие в управлении данными, потому что все аспекты нашей личной и профессиональной жизни теперь связаны с интернетом. Пандемия COVID-19 расширила связи между людьми, ведь многие из нас вынуждены были работать из дома.

¹ The New York Times описала последствия взлома учетных записей Marriott (<https://oreil.ly/0zJdf>).

Тенденции объединения массивных данных, более мощной продвинутой аналитики и более быстрого распространения вызвали множество споров, этических вопросов и дискуссий об опасностях в сфере данных. По мере того как компании будут совершать ошибки и пересекать границы этических норм, правительства, как мне кажется, ужесточат регулирование и потребуют большей безопасности, контроля и внимания. Мы изучили лишь часть истинной конфиденциальности и этических проблем, связанных с данными. Регулирование заставит большие компании быть честными в плане того, какие данные собирать, покупать, объединять, анализировать и распространять (продавать). Крупным компаниям придется задуматься о подходах, ориентированных на прозрачность и конфиденциальность, и о том, как решать серьезные вопросы регулирования.

Регулирование — сложный вопрос. Представьте себе ситуации, когда имеется несколько облачных окружений и разных сервисов SaaS, использующих распределенные данные. Удовлетворить требования европейского Общего регламента по защите данных (General Data Protection Regulation, GDPR) (<https://gdpr-info.eu/>) и Закона штата Калифорния о защите конфиденциальности потребителей (California Consumer Privacy Act, CCPA) (<https://oag.ca.gov/privacy/ccpa>) сложно, потому что компании должны иметь представление о персональных данных и контролировать их независимо от того, где они хранятся. Управление персональными данными и работа с ними — приоритет для многих крупных фирм¹.

Более строгие нормативные требования и этика данных в будущем приведут к дополнительным ограничениям и усилению контроля. Важнейшее значение приобретет понимание того, откуда данные пришли и как они распределяются. Потребуется более жесткое внутреннее управление. Тенденция к усилению контроля противоречит методикам быстрой разработки программного обеспечения, признаки которых — меньше документации и средств внутреннего контроля. Это приводит к появлению другой, оборонительной, точки зрения на то, как управление данными осуществляется внутри компаний. Большая часть этих проблем рассмотрена в главе 7.

Необходимо интегрировать операционные системы и системы обработки транзакций

Важность быстрого реагирования на события в деловом мире диктует новые задачи. Традиционно существует четкое разделение между приложениями

¹ Персональные данные — это любая информация, относящаяся к идентифицированному или идентифицируемому физическому лицу.

обработки транзакций (операционными) и аналитическими приложениями. Так происходит потому, что возможностей систем обработки транзакций обычно недостаточно для доставки больших объемов данных или их постоянного распространения. Лучшей практикой всегда считалось разделение стратегии данных на две части: 1) обслуживание транзакций и сохранение аналитических данных и 2) обработка больших данных.

В то же время это четкое разделение становится все менее очевидным. Ожидается, что *операционная аналитика*, которая построена вокруг прогнозирования и улучшения существующих операционных процессов, будет тесно взаимодействовать как с транзакционными, так и с аналитическими системами. Аналитические результаты необходимо снова интегрировать в ядро операционной системы, чтобы эта информация могла использоваться в операционном контексте.

Эта тенденция требует другой архитектуры интеграции, объединяющей операционные и аналитические системы. Это также требует интеграции данных для работы с разными скоростями: со скоростью операционных систем и со скоростью аналитических систем. В этой книге мы рассмотрим варианты сохранения исторических данных в *исходном операционном контексте*, делая их доступными одновременно для операционных и аналитических систем.

Для монетизации данных требуется экосистемная архитектура

Многие люди считают свое предприятие единой бизнес-экосистемой с четкими границами, но мнениям свойственно меняться¹. Компании все чаще интегрируют свои основные бизнес-функции и услуги со сторонними организациями и их платформами. Они монетизируют данные, делают API общедоступными и используют открытые данные в целом².

Как следствие, данные чаще распределяются между окружениями и становятся более децентрализованными. При этом используются облачные решения или SaaS. В итоге данные получаются раскиданными по разным местам, что затрудняет их интеграцию и управление. Кроме того, возникают проблемы с пропуск-

¹ Джеймс Мур (James Moore), автор книги *The Death of Competition* (Harper, 1997), определил бизнес-экосистему как «совокупность компаний, которые работают совместно и на конкурентной основе для удовлетворения потребностей клиентов».

² Открытые данные — такие, которые можно свободно использовать и сделать общедоступными. В McKinsey считают, что монетизация данных меняет способ ведения бизнеса (<https://oreil.ly/yNH8V>).

ной способностью сети, подключением и задержкой, когда данные не передаются с достаточной скоростью платформам или окружениям, где они применяются. Использование единой стратегии общедоступного облака не решит такие проблемы. Поэтому, если вам нужно, чтобы API и системы SaaS работали хорошо и использовали возможности общедоступного облака, придется освоить интеграцию данных — эта книга вам помочь.

Тенденции, о которых я рассказываю, важны и будут влиять на то, как люди используют данные и как компании должны организовывать свою архитектуру. Объемы данных увеличиваются все быстрее, вычислительные мощности растут, а аналитические методы развиваются. Миру нужно все больше данных, а значит, их надо быстро распространять и наладить более строгое управление данными. Управление должно быть децентрализованным из-за стремления шире использовать облачные вычисления, SaaS и микросервисы. Все эти факторы необходимо уравновесить, чтобы сократить время выхода на рынок и получить дополнительные преимущества в конкурентной борьбе. Эта рискованная комбинация заставляет нас совершенно иначе управлять данными.

Предприятия обременены устаревшими архитектурами данных

Одна из самых больших проблем, с которой сталкиваются многие предприятия, — это получение выгоды от существующих корпоративных архитектур данных¹. Большинство таких архитектур используют монолитную структуру — корпоративное хранилище или озеро данных — и централизованно управляют данными, распределяя их. В сильно распределенном окружении эти архитектуры не будут отвечать потребностям будущего. Рассмотрим некоторые их характеристики.

Корпоративное хранилище данных и бизнес-аналитика

Архитектуры данных первого поколения основаны на хранилищах данных и бизнес-аналитике. Идея заключается в том, что существует единое центральное интегрированное хранилище, содержащее подробные и стабильные данные, накопленные за годы существования организации. У такой архитектуры есть свои недостатки.

¹ Под архитектурой данных здесь понимаются инфраструктура и данные, а также схемы, интеграция, преобразования, хранилище и рабочий процесс, необходимые для выполнения аналитических требований информационной архитектуры.

Унификация корпоративных данных — очень сложный процесс, который занимает много лет. Высоки шансы, что в разных предметных областях, отделах и системах значение данных различается¹. Атрибуты данных могут иметь одинаковые имена, но разные значения и определения. Поэтому мы либо создаем много вариантов, либо просто принимаем различия и несоответствия. Чем больше данных мы добавляем и чем больше противоречий и несоответствий имеется в определениях, тем труднее будет их согласовать. Скорее всего, получится единый контекст, бессмысленный для всех. Для продвинутой аналитики вроде машинного обучения исключение контекста может стать большой проблемой, ведь на бессмысленных данных не сделать точный прогноз.

Корпоративные хранилища данных (enterprise data warehouses, EDW) ведут себя как *интеграционные базы данных* (рис. 1.1). Они действуют как хранилища для множества приложений, потребляющих данные. Это означает, что они являются связующим звеном для всех приложений, желающих получить доступ к хранилищу. Изменения необходимо вносить осторожно из-за множества зависимостей между различными приложениями. Некоторые изменения могут также вызвать «эффект ряби» для других изменений. В этом случае образуется большой ком грязи.



Рис. 1.1. Корпоративные хранилища данных обычно имеют множество точек сопряжения, этапов интеграции и зависимостей

Высокая сложность хранилища данных и тот факт, что им управляет одна центральная группа, плохо сказывается на гибкости. Увеличивается время ожидания, что заставляет людей искать обходные решения. Один разработчик, например, может обойти уровень интеграции и напрямую отобразить данные из промежуточного уровня в своем киоске данных. Другой создаст представление для быстрого объединения данных. Такой *технический долг* (будущая доработка) позже приведет к проблемам. Архитектура станет настолько сложной, что люди перестанут понимать общую структуру и обходные пути, созданные ради своевременной доставки.

¹ Предметная область — это область исследования, которая определяет набор общих требований, терминологии и функциональности для любого программного обеспечения, созданного для решения проблемы в области компьютерного программирования.

БОЛЬШОЙ КОМ ГРЯЗИ

Большой ком грязи (<https://oreil.ly/8bY2t>) — хаотично устроенные, раскидистые, неряшливые джунгли из спагетти-кода, скрепленные изолентой и проволокой. Этот популярный термин был впервые введен Брайаном Футом (Brian Foote) и Джозефом Йодером (Joseph Yoder). Большой ком грязи описывает системную архитектуру — монолитную, трудную для понимания и обслуживания и тесно связанную из-за множества зависимостей. На рис. 1.2 показана диаграмма зависимостей, иллюстрирующая подобную архитектуру (<https://oreil.ly/Fdckr>). Каждая линия представляет связь между двумя программными компонентами.

Как видите, в большом коме грязи каждый компонент связан со всеми остальными, из-за чего практически невозможно изменить один компонент, не затрагивая другой.

Хранилища данных с их уровнями, представлениями, бесчисленными таблицами, взаимосвязями, сценариями, заданиями и потоками планирования часто становятся причиной появления хаотичной паутины зависимостей. Это настолько сложно, что через некоторое время вы часто получаете большой ком грязи.

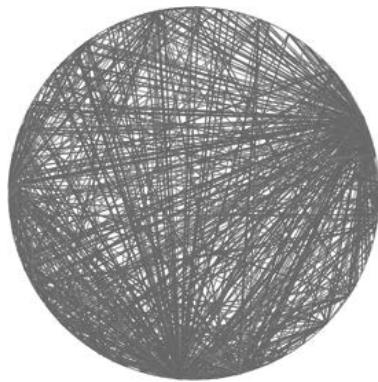


Рис. 1.2. Диаграмма зависимостей — это математическая модель, часто используемая в программной архитектуре для идентификации компонентов и функциональных единиц проекта

Хранилища данных тесно связаны с выбранным базовым решением или технологией. Это означает, что потребители, которым нужны другие шаблоны чтения, будут вынуждены экспорттировать данные в другие окружения. По мере того изменения рынка и появления баз данных новых типов хранилища становятся все более разбросанными, из-за чего приходится снова экспорттировать данные. Эта тенденция подрывает общее видение эффективного использования единого центрального репозитория и базового (дорогостоящего) оборудования.

Управление жизненным циклом исторических данных часто сопряжено со сложностями. Хранилища данных рассматриваются как архивы истины, по-

зволяющие операционным системам очищать ненужные им данные, зная, что они останутся в хранилище. Для продвинутой операционной аналитики, возникшей после появления хранилищ данных, это может быть проблемой. Если данные были преобразованы и больше не распознаются как пригодные для операционного использования, то быстро сделать их доступными будет непросто, учитывая, что многие хранилища обычно обрабатывают данные в течение многих часов.

Хранилища часто не имеют информации о разовом потреблении и дальнейшем распределении данных, особенно когда они передаются за пределы экосистемы. В соответствии с современным законодательством владение данными и понимание их потребления и распределения играют важную роль, потому что вы должны быть в состоянии объяснить, какие персональные данные были использованы, кем и с какой целью.

Качество данных тоже часто оставляет желать лучшего. Кому принадлежат данные в хранилищах? Кто несет ответственность, если исходные системы доставляют поврежденные данные? Я анализировал ситуации, когда инженеры сами решали вопросы качества данных. В одном случае они зафиксировали данные на промежуточном уровне, чтобы правильно загрузить их в хранилище. Эти исправления стали постоянными, и со временем пришлось применить сотни дополнительных сценариев, прежде чем можно было начать обработку данных. Эти сценарии не являются частью заслуживающих доверия процессов ETL и не позволяют отследить происхождение данных.

Замещающая миграция может стать рискованной и трудоемкой, если взять во внимание общий объем данных в хранилище, годы, которые потребовались на его разработку, человеческие знания и интенсивное использование в бизнесе. Поэтому многие предприятия продолжают использовать эту архитектуру и создавать на ее основе свои бизнес-отчеты, информационные панели и приложения, активно потребляющие данные¹.

Озеро данных

По мере роста объемов данных и необходимости более быстрого анализа инженеры начали работать над другими концепциями. *Озера данных* возникли как альтернатива для доступа к большим объемам сырых, необработанных данных².

¹ Информационные панели мониторинга более наглядны и используют различные типы диаграмм. Отчеты, как правило, имеют табличную форму, но могут содержать дополнительные диаграммы или их компоненты.

² Джеймс Диксон (James Dixon), бывший технический директор Pentaho, придумал термин «озеро данных» (https://oreil.ly/4E_WB).

Используя их, потребитель может сам решать, как использовать, преобразовывать и интегрировать эти данные.

Озера, как и хранилища, считаются централизованными (монолитными) и отличаются от последних тем, что хранят данные до их преобразования, очищения и структурирования. Поэтому схемы часто определяются при чтении данных. В этом состоит отличие от хранилищ, в которых используется предопределенная и фиксированная структура. Озера данных также обеспечивают большее разнообразие данных за счет поддержки нескольких форматов: структурированных, полуструктурных и неструктурных.

Хранилища и озера также сильно различаются в плане используемой базовой технологии. Первые обычно создаются с помощью RDBM, а вторые — с помощью распределенных БД или систем NoSQL. Часто используются общедоступные облачные сервисы. Недавно распределенные и полностью управляемые облачные базы данных¹ поверх контейнерной инфраструктуры упростили задачу масштабного управления централизованными репозиториями данных. При этом появились преимущества в виде большей гибкости и меньшей стоимости².

Многие озера собирают чистые, неизмененные, необработанные данные из исходных систем (рис. 1.3). Сохранение данных в первозданном виде — точных копий — выполняется быстро и обеспечивает мгновенный доступ к ним специалистам по анализу и обработке данных. Но с первозданными данными есть одна сложность: они требуют предварительной обработки. Приходится решать проблемы с качеством данных, их объединением и обогащением другими данными, чтобы они соответствовали контексту. Это добавляет много рутинной работы — еще одна причина, почему озера данных обычно объединяются с хранилищами. Хранилища в этой комбинации действуют как высококачественные репозитории очищенных и согласованных данных, в то время как озера выступают в роли (специальных) аналитических окружений, содержащих много необработанных данных для облегчения анализа.

Создавать озера данных, как и хранилища, непросто. Ник Худекер (Nick Heudecker), аналитик из Gartner, написал в «Твиттере», что, по его мнению, частота отказов внедрения озер данных превышает 60 %³. Попытки реализации

¹ Docker , Inc. — ведущая компания, создающая инструменты на основе Docker (<https://oreil.ly/15ZW9>), прекрасно объясняет концепцию контейнеров (<https://oreil.ly/jpyb8>).

² Эластичность — это степень, в которой системы могут адаптировать изменения своей рабочей нагрузки за счет автоматического выделения и удаления ресурсов.

³ TechRepublic даже заявляет, что 85 % всех проектов с большими данными оказываются неудачными (<https://oreil.ly/qKm0M>).

озера обычно терпят неудачу из-за их огромной сложности, проблем с обслуживанием и общих зависимостей.

- Данные, загружаемые в озеро, часто не обработаны и являются собой сложную совокупность различных представлений. В озере могут быть десятки тысяч таблиц, непонятные структуры данных и технические значения, понятные только самому приложению. Кроме того, данные тесно связаны с исходными системами, так как унаследованная структура является идентичной копией. К тому же существует реальная опасность нарушения работы конвейера загрузки необработанных данных, если их источники вдруг изменятся.
- Аналитические модели в озерах часто обучаются и на необработанных, и на согласованных данных. Не исключено, что инженеры и специалисты по обработке и анализу данных вручную собирают и создают данные, а также управляют конвейерами и моделями в рамках своих проектов. Как следствие, озера данных несут значительные (операционные) риски.
- Озера данных часто представляют собой единую платформу и используются во многих различных сценариях. Такие платформы очень сложно поддерживать из-за их тесной связанности, проблем совместимости, общих библиотек и конфигураций.

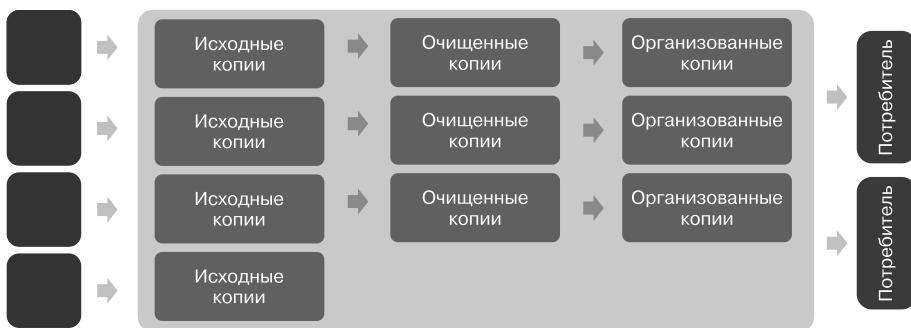


Рис. 1.3. Озера данных — это гигантские области хранения объектов, которые собирают необработанные данные из различных источников. Во многих случаях это просто пул таблиц без каких-либо определенных логических границ областей

Эти проблемы — лишь часть тех причин, по которым проваливается так много проектов, связанных с большими данными. К прочим можно отнести сопротивление руководства, внутреннюю политику, недостаток опыта, а также проблемы безопасности и управления.

Централизованное представление

Хранилища и озера данных можно масштабировать с помощью таких методов, как ELT на основе метаданных, виртуализация данных, облачные технологии, распределенная обработка, поглощение данных в реальном времени, машинное обучение для пополнения информации и т. д. Но есть гораздо более серьезная проблема: централизованное мышление, лежащее в основе этих архитектур. Здесь подразумеваются централизованное управление и владение данными, кластеризация ресурсов и использование централизованных моделей, предлагающих, что все должны использовать одинаковые термины и определения. У такой централизации есть еще один негативный эффект: убирая специалистов по данным из бизнес-областей, мы лишаемся бизнес-идей и творческих подходов. Команды вынуждены постоянно взаимодействовать друг с другом. Не зря современные технологические компании отстаивают *предметно-ориентированное проектирование* — подход к разработке программного обеспечения, впервые предложенный Эриком Эвансом (Eric Evans). Он включает лучшие общепринятые практики и стратегические, философские, тактические и технические элементы.

Итоги главы

Хранилища данных никуда не исчезнут, потому что всегда придется согласовывать данные из разных источников в рамках определенного контекста. Не исчезнут и шаблоны разделения посредством промежуточных данных, равно как и этапы очистки, исправления и преобразования схем. Любой архитектурный стиль моделирования Билла Инмона (Bill Inmon) (<https://oreil.ly/wIB9w>), Ральфа Кимбалла (Ralph Kimball) (<https://oreil.ly/ToDnb>) или моделирования хранилища данных (<https://oreil.ly/Fj8JL>) может подойти в зависимости от особенностей вариантов использования. То же самое относится и к озерам данных: потребность в распределенной обработке огромных объемов данных для аналитики исчезнет нескоро.

Но мы должны подумать о том, как следует в целом управлять данными и распространять их. Большие разрозненные хранилища вроде корпоративных исчезнут из-за невозможности масштабирования. Тесно связанные уровни интеграции, потеря контекста и рост потребления данных заставят компании искать альтернативы. Еще одна крайность — архитектуры озер данных, которые извлекают данные в первоначальном виде. Необработанные данные, которые могут измениться в любой момент, не позволят экспериментам и сценариям использования когда-нибудь внедрить их в производство. Работа над такими данными рутинна и скучна.

Масштабируемая архитектура

Решением проблем разрозненных данных может стать масштабируемая архитектура: типовая и предметно-ориентированная архитектура с набором схем, проектов, принципов, моделей и наилучших практик, которая упрощает и интегрирует управление данными во всей организации с помощью распределения. Я вижу это как архитектуру, которая сближает все области управления данными, предлагая единый механизм управления безопасностью, основными данными, метаданными и моделированием данных; архитектуру, которая может работать с использованием комбинации нескольких облачных провайдеров и локальных платформ, но при этом дает необходимые контроль и гибкость. Она упрощает работу команд, предоставляя не зависящие от предметной области и повторно используемые архитектурные блоки, и при этом обеспечивает гибкость, сочетая различные стили доставки данных с разнообразными технологиями. Масштабируемая архитектура позволяет командам самостоятельно превращать данные в ресурс без помощи центральной группы.

Масштабируемая архитектура, описанная в этой книге, состоит из обширного набора принципов управления данными. Она потребует от вас, например, идентифицировать и классифицировать подлинные и уникальные данные, определить их качество в источнике, обеспечить поддержку метаданных и провести четкие границы. Следуя этим принципам, предприятия дадут своим командам возможность быстро распределять и использовать данные, сохраняя свою независимость. Эта архитектура также сопровождается моделью управления: инженеры должны научиться создавать хорошие абстракции и конвейеры данных, а владельцы бизнес-данных — нести ответственность за эти данные и их качество, обеспечивая ясность контекста.

ГЛАВА 2

Введение в масштабируемую архитектуру: организация данных в масштабе

Какая архитектура нужна предприятию, чтобы управление им основывалось на данных? Как эффективно распределять данные, сохраняя гибкость, безопасность и контроль? В этой главе мы рассмотрим эти вопросы и заложим основу для управления данными.

Современные тренды толкают нас на переосмысление способов управления данными и их интеграции. Ранее мы разобрали тесную связь, возникающую при создании точных копий данных, и трудности практического анализа необработанных данных. Мы также обсудили проблемы унификации и огромные усилия, которые прилагаются к созданию интегрированного хранилища данных, и их влияние на гибкость. Нам необходимо перейти от объединения всех данных в одном хранилище к подходу, который позволяет предприятиям, командам и пользователям легко и безопасно распределять, собирать и использовать данные. Платформы, процессы и шаблоны должны упрощать работу другим. Нам нужны простые, хорошо документированные, быстрые и легкие в использовании интерфейсы. Нам нужна масштабируемая архитектура управления данными. Именно эти вопросы мы и затронем в главе. А начнем мы с того, как организовать ландшафт и интегрировать данные.

Крупномасштабная архитектура в моем представлении ориентирована на управление данными и их интеграцию. Это архитектура для предприятий, которая позволяет командам безопасно и легко предоставлять данные, сохраняя гибкость и контроль. Как и во многих других архитектурах, в ней используются *архитектурные строительные блоки*, которые относятся к «пакету функций, определенных для удовлетворения потребностей бизнеса»¹. Эти блоки будут

¹ Согласно OpenGroup (<https://oreil.ly/YXEbm>) способ объединения функциональности, продуктов и нестандартных разработок в архитектурные блоки зависит от архитектуры.

использоваться снова и снова, чтобы помочь вам понять, какую именно часть архитектуры мы обсуждаем.

Начнем с традиционных принципов определения наиболее важных архитектурных блоков. Далее мы обсудим дополнительную литературу и сделаем выводы. Затем, наконец, рассмотрим новую архитектуру и ее обоснование, а также раскроем, что находится внутри нее. К концу этой главы вы поймете, как различные архитектурные блоки работают и связаны между собой. Эта базовая теория необходима для понимания основных движущих сил новой архитектуры. В следующих главах мы подробнее рассмотрим шаблоны, проекты, диаграммы и рабочие процессы, и вы начнете понимать, как эта архитектура связывает воедино все области управления данными.

Общепризнанные отправные точки

Прежде чем погрузиться в обсуждение, я хочу выделить главные отправные точки. Они формируют архитектуру и содержат элементы, на которые я часто ссылаюсь.

У каждого приложения есть база данных

Приложения тесно связаны с базами данных. В контексте управления данными и их перемещения между приложениями мы можем допустить, что у каждого приложения всегда есть база данных. То есть всякий раз, сталкиваясь с необходимостью потребления данных, вы должны хранить их в БД приложения. Да, бывают приложения, хранящие свои данные в другом месте, а еще приложения, использующие одну и ту же базу данных, или приложения, выполняющие обработку в памяти. Но всем этим приложениям все равно нужно где-то хранить свои данные. Так что у них всегда есть хранилище в той или иной форме, а значит, и хранилище данных приложения.

Приложения специфичны и обладают уникальным контекстом

В главе 1 я отметил, что приложения используются для решения *конкретных* задач. Данные каждого приложения уникальны. Существует несколько этапов проектирования и разработки приложений. Все начинается с определения концепции и разработки проекта; затем мы трансформируем наши знания в логическую модель данных, определяющую абстрактную структуру концептуальной информации и требований. Наконец, мы создаем физическую модель данных приложения: истинный проект приложения и базы данных. Физическая модель данных уникальна и принимает как контекстные, так и нефункциональные требования к тому, как приложение и БД будут спроектированы и использованы.

Золотой источник

В фрагментированной и распределенной среде порой сложно определить авторитетные источники исходных и уникальных данных. Поэтому важно знать, откуда поступают данные и где ими управляют. В этой книге я обсуждаю основополагающие концепции золотого источника и золотого набора данных¹.

- *Золотой источник* — это авторитетное *приложение*, в котором все аутентичные данные обрабатываются в определенном контексте. Он состоит из одного или нескольких золотых наборов данных².
- *Золотой набор данных* — это созданные надежные оригинальные *данные*. Он подлинный и уникальный и должен быть точным, полным и известным³. Золотой набор данных состоит из элементов данных (<https://oreil.ly/RGKhT>) — элементарных единиц удобочитаемой информации, имеющих точное значение или точную семантику. У них есть определяемое имя, и они служат связующим звеном для других субъектов управления данными, таких как распоряжение данными.

Благодаря золотому источнику и золотому набору данных вы всегда сможете точно и последовательно отчитаться о своих данных. Это важно для надежного управления данными, о чем мы поговорим позже, в главе 7.

Дileммы интеграции данных не избежать

Для перемещения данных между приложениями их всегда приходится интегрировать. Это происходит из-за уникального контекста, в котором эти данные создаются. Неважно, что вы выполняете — ETL (extract, transform, and load — «извлечение, преобразование и загрузка») или ELT (extract, load, and transform — «извлечение, загрузка и преобразование»), виртуальным или физическим способом, пакетами или в реальном времени: от дилеммы интеграции данных никуда не деться. Преобразование всегда требуется при переносе данных из одного контекста в другой. Ключевое слово *всегда* образует новую архитектуру.

¹ W3C определяет набор данных (<https://oreil.ly/nvHnG>) как «коллекцию данных, которые можно получить или загрузить в одном или нескольких форматах. Неточное определение набора данных было сделано намеренно, чтобы созданный концептуальный набор данных можно было использовать в различных контекстах».

² Поиск надежных приложений порой непрост. Некоторые надежные источники могут быть скрыты за сложными шаблонами, которые необходимо понять, прежде чем найти, что вам нужно. Для получения дополнительной информации о золотых источниках см.: *Graham A. Mastering Your Data* (Koios, 2015).

³ Бывают исключительные ситуации, когда золотые наборы данных не управляются системой, которая фактически создала исходный фрагмент данных.

Приложения играют роли поставщиков и потребителей данных

Приложения выступают в роли либо поставщиков, либо потребителей данных, а иногда, как мы увидим, и в той, и в другой одновременно. Одно приложение использует данные, а другое создает и предоставляет их. В контексте интеграции данных во множестве приложений и систем это важно понимать. Роли поставщика и потребителя данных станут строительными блоками формальной архитектуры.

В зависимости от того, действует ли система или приложение в качестве поставщика или потребителя, правила игры меняются: применяются разные архитектурные принципы (рис. 2.1).



Рис. 2.1. Роли приложения как поставщика и потребителя данных будут формировать нашу архитектуру

Роли поставщика и потребителя данных также применимы к *внешним сторонам*. Внешними называют стороны, действующие за пределами логических границ экосистемы предприятия. Обычно они находятся в отдельных неконтролируемых местах в сети. В главе 1 мы говорили, что компании рассматривают общедоступную сеть как кладезь (открытых) данных, которые они могут монетизировать для создания услуг и обмена ими. Этот новый способ сотрудничества изменил границы организаций. Новая архитектура, которую я представляю, требует гибкости, чтобы внешние стороны могли быть как поставщиками, так и потребителями данных. Обычно ради этого приходится применять дополнительные меры безопасности, ведь внешние стороны не всегда надежны или известны. Иногда они напрямую связаны с более широким контекстом.

Уникальный контекст, в котором работают приложения, роли поставщика и потребителя данных, а также преобразование, которое всегда происходит между приложениями, будут определять новую архитектуру. Но что еще нужно учитывать? Что делает общую архитектуру хорошей? В следующих разделах мы исследуем эти вопросы.

Основные теоретические соображения

Интеграция данных между приложениями — это управление сложностью взаимодействия и согласования данных между системами. В архитектуре предприятия мы обычно рассматриваем более широкую картину. Но как компоненты работают вместе для интеграции данных внутри приложения и что мы можем из этого извлечь?¹

Интеграция приложений находится на уровне архитектуры или разработки программного обеспечения. На абстрактном уровне интеграция корпоративных данных и интеграция программного обеспечения близки, а порой и частично совпадают (рис. 2.2).



Рис. 2.2. Дисциплины интеграции данных и интеграции программного обеспечения имеют множество пересечений

Далее мы внимательно рассмотрим несколько шаблонов разработки программного обеспечения, позволяющих разобраться в повторяющихся проблемах. Их понимание помогает командам разработчиков избежать создания зависимостей и сложностей.

Принципы объектно-ориентированного программирования

Упомянутые ниже элементы — это те принципы объектно-ориентированного проектирования, которые часто используются для создания независимых программных компонентов. Они все еще применяются в современных популярных методах.

¹ Архитектура предприятия (enterprise architecture, EA) — это план воплощения стратегии предприятия (бизнес-целей и задач) в успешные изменения. Она логически фокусируется на целом предприятии, включая бизнес, информацию (данные), приложения, безопасность и инфраструктуру, тогда как архитектура данных в первую очередь ориентирована на данные.

Первый принцип, который мы рассмотрим, — это *объектно-ориентированное программирование* (ООП). Управление сложностью приложения и кода в целом осуществляется путем абстрагирования сложной логики, реализации общих функций для выполнения повторяющихся задач и создания общих интерфейсов для других компонентов.

В брошюре Роберта К. Мартина (Robert C. Martin) *Design Principles and Design Patterns* (object-mentor.com, 2000) изложены идеи, идеально подходящие для интеграции данных. Многие из его принципов проектирования до сих пор применяются в стандартной практике.

- *Принцип единственной ответственности.* Этот принцип заключается в задании четких обязанностей и границ. Как пишет Мартин, «этот принцип — для людей. Нужно изолировать модули от сложностей организации в целом и спроектировать системы так, чтобы каждый модуль отвечал за потребности только одной бизнес-функции»¹.
- *Принцип инверсии (внедрения) зависимостей.* Он гласит: «Модули высокого уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракций. Абстракции не должны зависеть от деталей — детали должны зависеть от абстракций»². Речь идет о сокрытии внутренней сложности и деталей. Абстракция более стабильна, чем лежащая в основе логика.
- *Принцип открытости/закрытости.* Этот принцип гласит, что «модуль должен быть открыт для расширения, но закрыт для модификации»³. Когда мы изменяем функцию, все, что зависит от нее, также должно быть изменено. Нельзя, чтобы изменения в обмене данными вызывали каскад последующих изменений в зависимых системах или приложениях.
- *Принцип устойчивых зависимостей.* Принцип гласит, что «программные модули зависят от направления устойчивости. Устойчивость связана с объемом работы, необходимой для внесения изменений»⁴. Функциональность приложения должна основываться только на функциях, которые по крайней мере столь же устойчивы, как и модуль.
- *Принцип устойчивой абстракции.* Он гласит, что «компонент должен быть настолько абстрактным, насколько и стабильным»⁵. Преимущество абстрактных стабильных компонентов в том, что вы их легко расширять, не нарушая проект.

¹ Мартин Р. К. и др. Быстрая разработка программ. Принципы, примеры, практика.

² Там же.

³ Martin R. C. The Open-Closed Principle, C++ Report. 1996. <https://oreil.ly/dIJgo>.

⁴ Мартин Р. К. и др. Быстрая разработка программ. Принципы, примеры, практика.

⁵ Martin R. C. OO Design Quality Metrics. 1994. <https://oreil.ly/jl8Cq>.

Новая архитектура была вдохновлена именно этими принципами Мартина, потому что способ взаимодействия приложений через интерфейсы напоминает способ взаимодействия компонентов внутри приложения. Чтобы избежать нарушений в работе приложений и систем при каждом изменении интерфейса, мы должны действовать в соответствии с принципами Мартина. Они пользуются большим уважением и повлияли на многие структуры и методологии разработки. Одна из них называется *предметно-ориентированным проектированием* (domain-driven design, DDD).

Предметно-ориентированное проектирование

Предметно-ориентированное проектирование — это подход к разработке программного обеспечения, который включает сложные системы для крупных организаций, первоначально описанный Эриком Эвансом (Eric Evans)¹. Принцип DDD популярен, потому что многие из его высокогоуровневых практик оказали влияние на современные подходы к разработке ПО и приложений, например микросервисы.

Ограниченный контекст

Один из шаблонов предметно-ориентированного проектирования называется *ограниченным контекстом*. Ограничные контексты используются для установки логических границ пространства решений предметной области, чтобы лучше управлять сложностью. Важно, чтобы команды понимали, какие аспекты, включая данные, они могут изменять самостоятельно, а какие являются общими зависимостями, изменения в которых необходимо согласовывать с другими командами, чтобы ничего не сломать. Границы помогают командам и разработчикам более эффективно управлять зависимостями.

Логические границы, как правило, явные и применяются в областях с четкой и более выраженной связностью. Эти зависимости предметной области могут располагаться на разных уровнях, таких как определенные части приложения, процессы, связанные структуры баз данных и т. д. Ограниченный контекст является полиморфным и может применяться ко многим различным точкам зрения. *Полиморфизм* означает возможность изменения размера и формы ограниченного контекста в зависимости от точки зрения и окружения. Это также означает, что ограниченный контекст нужно использовать явно; иначе цель его применения останется довольно неясной.

¹ Эванс Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем.

ПРЕДМЕТНЫЕ ОБЛАСТИ И ОГРАНИЧЕННЫЕ КОНТЕКСТЫ

DDD различает ограниченные контексты, предметные области и подобласти. *Предметные области* — это зоны проблем, которые мы пытаемся решить; те области, в которых сочетаются знания, поведение, законы и действия; те области, в которых мы видим семантическую связь: поведенческие зависимости между компонентами или службами. Предметные области обычно разбиваются на подобласти ради более эффективного управления сложностью. Типичный пример — разбиение предметных областей так, чтобы каждая подобласть соответствовала отдельному подразделению организации.

Не все подобласти одинаковы. Их можно разделить на основные, общие или вспомогательные. *Основные подобласти* являются наиболее важными. Это секретный ингредиент, который делает бизнес уникальным. *Общие подобласти* неспецифичны и обычно легко управляются с помощью готовых продуктов. *Вспомогательные подобласти* не дают конкурентных преимуществ, но необходимы для работы организации. Обычно они довольно простые.

Ограниченные контексты — это логические (контекстные) границы. Они сосредоточены на пространстве решений: проектировании систем и приложений. Именно в этом месте разумнее всего сфокусироваться на пространстве решений. Это может быть код, дизайн базы данных и т. д. Области и ограниченные контексты могут совпадать, но их не обязательно связывать. Ограниченные контексты носят технический характер и поэтому могут охватывать несколько предметных областей и подобластей.

Идея заключается в том, что после задания логических границ зоны ответственности становятся более явными и управляемыми. Общение между членами команды становится эффективнее, ведь все они работают над схожими задачами. Установление логических границ ПО похоже на принцип *единственной ответственности*, который гласит: «Что принадлежит одной области, должно оставаться (и эффективно управляться) в этой же области».

На уровне предприятия мы логически группируем приложения с высокой степенью связности или общими интересами. Общие интересы обычно находятся на уровне связности задач и ответственности бизнеса. Некоторые называют их функциональными областями, логическими группами, кластерами или организационными возможностями. Идея группировки сущностей, процессов или приложений не нова.

Модель предметно-ориентированного проектирования отличается от «традиционной» группировки *строгими границами* DDD. Эванс утверждает, что ограниченные контексты могут развиваться независимо, но должны быть разделены. Разделение обычно осуществляется через сокрытие сложных внутренних функций приложения и обеспечение определенного уровня устойчивости интерфейсов и уровней. Эти принципы аналогичны принципам *инверсии зависимостей* и *устойчивых зависимостей*.

Единый язык

«Единый язык — термин, который Эрик Эванс использует в предметно-ориентированном проектировании для описания практики построения общего строгого языка для общения разработчиков и пользователей», — сказал Мартин Фаулер (Martin Fowler) (<https://oreil.ly/enG7A>). Этот язык похож на определения, лексику или специализированную терминологию специалистов конкретной индустрии. Единый язык помогает сплотить людей в более крупных командах, потому что в большой команде или на предприятии часто бывает непросто прийти к взаимопониманию по вопросу языка. Чтобы избежать чрезмерного перекрестного общения и нестыковок в терминологии, для поддержки разработки приложений или программного обеспечения в предметной области используются единые языки. Унифицированного языка не существует, хотя между ними могут быть совпадения.



Я настоятельно рекомендую прочесть книгу *Semantic Software Design* Эбена Хьюитта (Eben Hewitt) (O'Reilly, 2019). В ней используется методология конструктивного мышления и проводится параллель между семантикой и проектированием архитектуры.

Ограниченный контекст и единый язык сильно взаимосвязаны, поскольку в DDD ожидается, что каждый ограниченный контекст будет иметь свой единый язык. Приложения или его компоненты, принадлежащие друг другу и управляющиеся в ограниченном контексте, должны использовать одинаковый язык. Если ограниченный контекст растет и нет взаимопонимания между членами команды, контекст может быть разбит на более мелкие части. Если ограниченный контекст изменится, ожидается, что и единый язык будет другим. Эмпирическое правило заключается в том, что один ограниченный контекст управляется одной командой (гибкой разработки или DevOps), потому что членам одной команды легче понимать текущую ситуацию и все ее зависимости.

Говоря об ограниченном контексте, я часто сравниваю его с культурой, чтобы показать, что в разных командах используются разные определения и терминология. Контекст специфичен и обычно основан на наших знаниях. У каждого ограниченного контекста своя цель, свой фон и, что наиболее важно, своя культура. Культура во многом определяет, как мы думаем, проектируем и моделируем. Внутри культуры есть субкультуры: группы людей в рамках более широкой культуры, которых объединяет общий набор целей и практик. То же можно сказать о предметно-ориентированном проектировании. Ограниченный контекст может состоять из нескольких подобластей или нескольких ограниченных контекстов. Точно так же в главе 1 мы говорили, что проект и модель данных приложения получают контекст из области, в которой они используются. Все эти подходы к проектированию тесно связаны.

Подход к предметно-ориентированному проектированию использует ограниченные контексты для установления границ независимых областей с более высоким уровнем связности. Если мы спроектируем DDD на нашу среду приложений на уровне предприятия, то границы области будут удерживать вместе не только приложение, но и язык, знания, ресурсы команды и технологии. Как показано на рис. 2.3, мы ожидаем, что каждый ограниченный контекст будет иметь свои приложение, данные, процессы и определения контекста (единый язык).

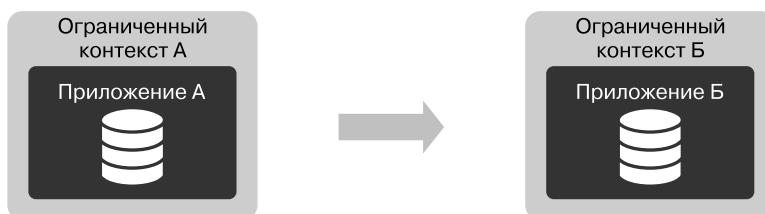


Рис. 2.3. DDD делит сложный ландшафт на ограниченные контексты и защищает границы между ними. В этом примере приложения А и Б имеют собственный ограниченный контекст

Недостатком использования приложений в качестве границ является то, что это все еще звучит очень размыто, если мы говорим о предприятии. Что объединяет элементы и компоненты в большую архитектуру? Обычно проблема заключается в том, как проектируются и разрабатываются приложения для бизнеса. Это подводит нас к следующей методологии: бизнес-архитектуре.

Бизнес-архитектура

Хорошо спроектированная *бизнес-архитектура* имеет решающее значение для успешного построения архитектуры предприятия. Гильдия бизнес-архитектуры описывает ее как «целостное, многомерное бизнес-представление о: возможностях, комплексной доставке ценностей, информации и организационной структуре, а также отношения между этими бизнес-взглядами и стратегиями, продуктами, политиками, инициативами и заинтересованными сторонами»¹. Архитекторы используют бизнес-архитектуру как основу для определения принципов, рекомендаций, желаемых результатов и границ предприятия и его бизнес-экосистемы. Бизнес-архитектура поддерживается путем построения целостных и многомерных бизнес-представлений с бизнес-возможностями². Мы можем

¹ Гильдия бизнес-архитектуры, статья: A Guide to the Business Architecture Body of Knowledge 7.5. – 2019. – С. 2. <https://oreil.ly/b3db5>.

² Определение возможности было первоначально придумано Ульрихом Хоманном (Ulrich Homann) в статье: A Business-Oriented Foundation for Service Orientation. 2006. Гильдия бизнес-архитектуры заимствовала его в проекте Business Architecture Body of Knowledge.

проводить параллель между ограниченным контекстом и бизнес-архитектурой или бизнес-возможностями. Еще один способ использования ограниченного контекста — посмотреть на него через призму бизнес-архитектуры.

Бизнес-возможности

Для решения бизнес-задач и удовлетворения потребностей часто используются бизнес-возможности, включающие создание объектов для каждой бизнес-цели. Бизнес-возможности — это строительный блок, используемый в бизнес-архитектуре. По словам Ульриха Хоманна (Ulrich Homann), бизнес-возможности — это «особые способы или возможности, которыми бизнес может обладать или обмениваться для достижения определенных результатов»¹. Бизнес-возможности — абстракция бизнес-реальности для помощи компаниям в достижении своих стратегических бизнес-целей и стремлений. Они фиксируют и описывают взаимосвязь между данными, процессами, организацией и технологиями в определенном контексте.



Модель бизнес-возможностей представляет общие стратегические бизнес-цели и действия организации в структурированном виде. Каждая бизнес-возможность реализуется как минимум единожды. Эту модель также можно использовать для сопоставления и построения графиков зависимостей. Например, сопоставление основных показателей эффективности управления данными с реализованными бизнес-возможностями показывает эффективность управления данными и его влияние на организацию.

Ограничные контексты, которые используются для задания логических границ, могут быть согласованы с бизнес-архитектурой. На самом высоком концептуальном уровне мы сопоставляем все стратегические бизнес-цели с бизнес-возможностями и группируем их вместе в бизнес-возможности и *потоки создания ценности*². Макет, более конкретная архитектура, создается уровнем ниже в архитектуре приложения. В этом проекте можно провести логические границы (решения), рассматриваемые как ограниченный контекст, представляющий функциональную и прикладную часть бизнеса и реализацию бизнес-архитектуры. Приложения и их компоненты используются для реализации конкретного *технологического* аспекта бизнес-возможностей.

¹ Крис Ричардсон (Chris Richardson), один из авторов Microservices.io, также признал точку зрения бизнес-возможностей (<https://oreil.ly/wBZj2>).

² Потоки создания ценности (<https://oreil.ly/ZgjuE>) — артефакты в рамках бизнес-архитектуры, которые позволяют бизнесу определять ценностное предложение, полученное от внешней (например, покупателя) или внутренней заинтересованной стороны организации.

ГРАНИЦЫ ПРЕДМЕТНОЙ ОБЛАСТИ И ДЕТАЛИЗАЦИЯ

Установка точных границ и детализации — неточная наука. Это мастерство, которое приходит с практикой. Для целей управления данными и понимания масштаба и сложности предприятия я предпочитаю проводить границы предметной области по логическим границам бизнес-архитектуры. Другие больше обращают внимание на организационные границы, бизнес-процессы или знания в предметной области. Еще один способ разграничения областей — разработать подробный план архитектуры программного обеспечения и определить, какие из компонентов приложения должны быть связаны сильнее или слабее. Этот метод особенно популярен у разработчиков микросервисов. Все эти подходы допустимы, и конкретный выбор будет зависеть только от контекста.

Чтобы лучше понять, что такие границы предметной области и как установить ограниченные контексты, рекомендую вам ознакомиться с историями о предметных областях (<https://oreil.ly/Imivx>), со штурмом событий (<https://oreil.ly/DH4OJ>) и с холстом ограниченного контекста (<https://oreil.ly/cWXvT>). Все эти методы предназначены для понимания сложности предметной области, изучения происходящего внутри нее и изучения возможностей ее структурирования или декомпозиции.

Бизнес-возможности в бизнес-архитектуре остаются абстрактными и могут быть реализованы несколько раз. При создании они называются *экземплярами возможности*. Чтобы помочь вам лучше понять эту концепцию, я сделал организационный пример (рис. 2.4) бизнес-возможностей и соответствующих экземпляров возможностей.

Управление взаимоотношениями с клиентами (customer relationship management, CRM) может быть реализовано в качестве бизнес-возможности как для розничного, так и для корпоративного бизнес-отдела компании. Эта же бизнес-возможность может быть осуществлена централизованно в виде модели обслуживания и предоставлена нескольким отделам. Интегрировать или централизовать — вопрос выбора. Централизованное внедрение CRM предпочтительнее, когда необходим контроль, например, с точки зрения безопасности или соответствия. Внедрение CRM в каждом бизнес-подразделении предпочтительнее при разной динамике или конфликте интересов команд.



В книге *Enterprise Architecture As Strategy* Джини В. Росс (Jeanne W. Ross), Питера Вейля (Peter Weill) и Дэвида С. Робертсона (David C. Robertson) (Harvard Business Press, 2006) упоминаются четыре различные классификации операционных моделей, которые также могут использоваться для бизнес-возможностей.

- *Унификация* — для высокой стандартизации и высокой интеграции.
- *Копирование* — для высокой стандартизации и низкой интеграции.
- *Координация* — для низкой стандартизации и высокой интеграции.
- *Диверсификация* — для низкой стандартизации и низкой интеграции.

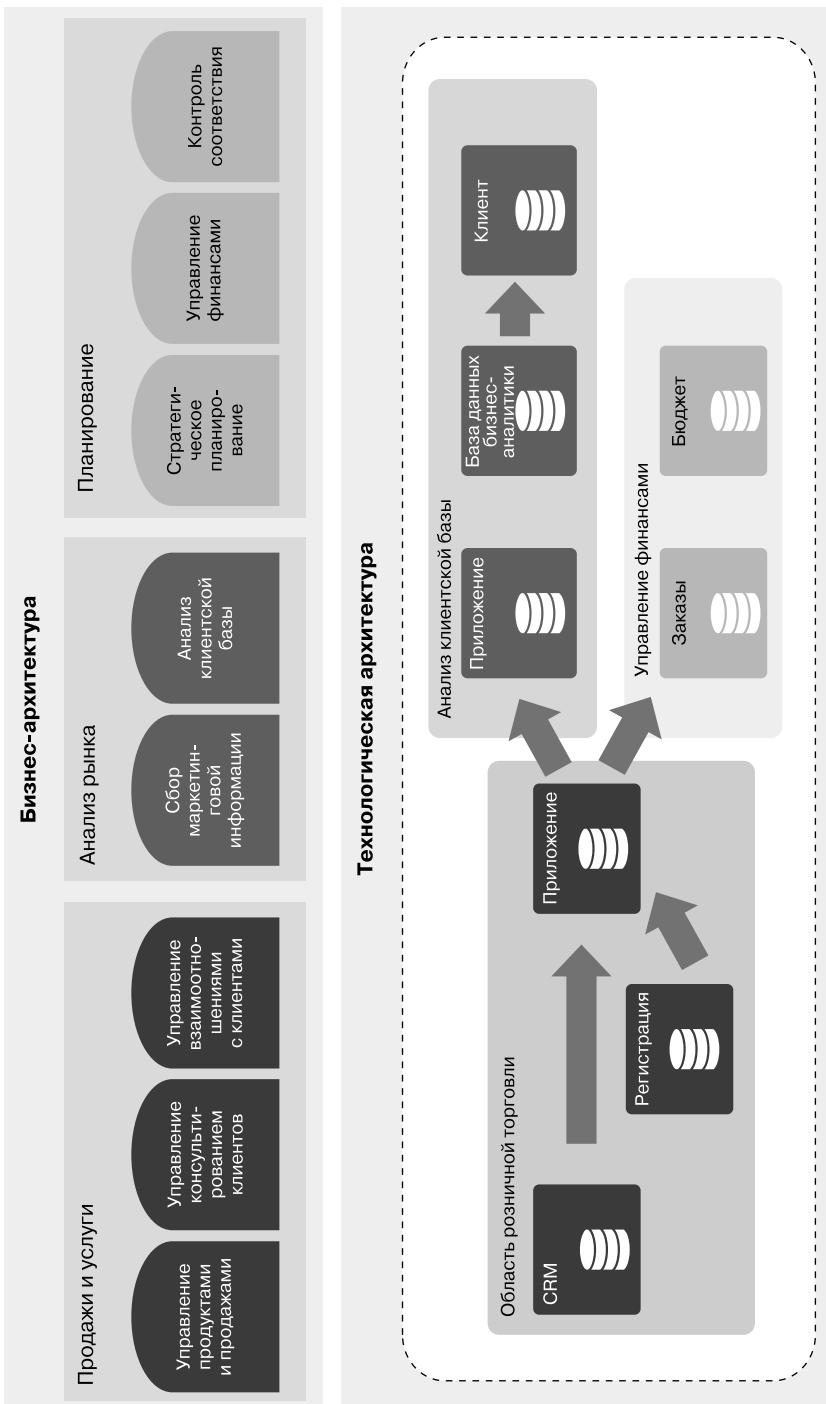


Рис. 2.4. Бизнес-возможности бизнес-архитектуры (верху) сосредоточены на предметном пространстве, а экземпляры возможностей (низи) — на пространстве решений. Политики макеты архитектуры приложения из экземпляров возможностей соответствуют границам бизнес-возможностей

Бизнес-возможности ничего не говорят о структуре и границах организации, а также о том, где разрабатывались возможности — внутри компании или за ее пределами. Они могут независимо перемещаться внутри или между организациями. Например, разработка сервиса онлайн-платежей изначально могла производиться внутри компании, а затем ее передали в PayPal. Такое организационное изменение не должно повлиять на возможности бизнеса, ведь конкретные возможности обработки онлайн-платежей для достижения целей предприятия остаются неизменными.

Связывание бизнес-возможностей с приложениями

Бизнес-возможности, экземпляры возможностей, ограниченные контексты и приложения могут быть согласованы. При этом важно соблюдать ряд основных правил.

Очень важно, чтобы бизнес-возможности оставались на бизнес-уровне и были абстрактными. Они представляют, чем занимается организация, и нацелены на *предметное пространство*. При осуществлении бизнес-возможности создается реализация — экземпляр возможности — для определенного контекста. В рамках этих границ в *пространстве решений* несколько приложений и компонентов могут работать вместе, обеспечивая определенную ценность для бизнеса. Приложения и компоненты, связанные с конкретными бизнес-возможностями, остаются отделенными от приложений, связанных с другими бизнес-возможностями.

Важно отметить, что логические границы приложения остаются неизменными: когда приложения разрабатываются самостоятельно, они связаны только с одним экземпляром возможностей. Они не должны охватывать несколько экземпляров возможностей или связываться с ними одновременно. Весь набор этих приложений в пределах экземпляра возможности образует ограниченный контекст. То же относится к данным и процессам — они тоже взаимосвязаны.

Для наших целей мы можем объединить две точки зрения, чтобы создать исчерпывающую классификацию. Само по себе понятие приложения остается абстрактным. Это логическая граница бизнеса, в которой можно создавать или развертывать *компоненты приложения*. Компоненты приложения, в отличие от приложений, материальны: они имеют размер, состав и существуют в контексте технологии. Например, модуль приложения, упакованный и развернутый как файл JAR, является вполне осозаемым артефактом.

Сочетание двух разных точек зрения очень ценно. На уровне приложений мы можем связать бизнес-возможности, владельцев продуктов, команды DevOps, проекты и т. д. На уровне компонентов приложения мы можем подключать

ИТ-продукты, версии, элементы конфигурации¹, контракты, планы обслуживания и многое другое.

Определение четких взаимосвязей между бизнес-возможностями и архитектурой приложения дает много преимуществ. Это значительно упрощает разделение интерфейсов и приложений и помогает лучше управлять данными. Вы можете утверждать, что все данные, которые создаются в определенном контексте для конкретной бизнес-возможности, принадлежат определенному владельцу продукта. Владение продуктом и владение данными внезапно становятся более согласованными. Вам также станет легче делать выводы об уровне возможностей, потому что приложения явно связаны с ними.

ГРАНИЦЫ ПРИЛОЖЕНИЯ

Что определяет область видимости и границы приложения? Определение границ приложения зависит от точки зрения, потому что слово *приложение* для разных людей имеет разное значение. С одной стороны отдельным приложением можно считать данные, которые упаковываются и развертываются вместе, независимо от других приложений. С другой стороны, приложениями могут считаться виртуальные адресные пространства, такие как виртуальные машины, где выполняются различные процессы. С третьей стороны, приложением можно считать интегрированный код, использующий общее окружение времени выполнения языка (<https://oreil.ly/NFQjS>) и взаимодействующий с другим интегрированным кодом посредством служб.

С точки зрения бизнеса приложением может считаться весь код и данные, которые используются вместе для удовлетворения конкретной бизнес-потребности. Такая точка зрения может определять границы между различными физическими окружениями, окружениями времени выполнения и платформами. Границы также могут быть основаны на (политическом) владении и организационной структуре компании. Если вкратце — интерпретация понятия приложения во многом зависит от вашей точки зрения.

При создании макетов и связывании бизнес-возможностей и их экземпляров с ограниченными контекстами несколько приложений могут работать вместе для удовлетворения одних и тех же бизнес-потребностей. На рис. 2.4 можно видеть, как для достижения стратегических целей CRM три приложения тесно взаимодействуют друг с другом. Вокруг или внутри этих приложений мы проводим ощущимые границы применимости, области видимости и ответственности. Приложения или компоненты приложений в пределах логической границы пространства решений области должны использовать единый язык и иметь возможность общаться напрямую. Они должны быть более тесно связаны; но за рамками ограниченного контекста может находиться другая предметная область — и тогда

¹ В терминологии библиотеки инфраструктуры информационных технологий (Information Technology Infrastructure Library, ITIL) элементы конфигурации (configuration items, CI) — это компоненты инфраструктуры, находящейся под управлением конфигурацией.

сработают тревожные сигналы: для сохранения гибкости в ландшафте наших приложений требуется разделение.



Некоторые люди используют термин «бизнес-сервисы»¹: логические группы операций, определенных в сервис-ориентированной архитектуре, представляющие бизнес-логику. Экземпляр возможности — это логическая граница, в которой находится реализация бизнес-возможностей. Бизнес-сервис — это деталь реализации: сервисы используются как стандартизованный метод связи.

Связывая разные модели, мы можем установить четкие принципы. Ограниченные контексты выводятся из бизнес-возможностей и сопоставляются исключительно с ними. Если изменяются возможности бизнеса, изменяются и ограниченные контексты. Вы уже узнали, что ограниченные контексты не связаны и взаимодействуют через интерфейсы или уровни. Это означает, что бизнес-возможности также должны быть разделены. Приложения, данные и процессы, не представляющие одну и ту же конкретную бизнес-возможность, не могут напрямую взаимодействовать с приложениями из другой бизнес-возможности и должны скрывать свою внутреннюю логику при общении с другими приложениями. Давайте разберем эту идею на конкретном примере.

Бизнес-архитектура в деле: пример из практики

Предположим, что юридическая фирма определила три четкие бизнес-цели и сопоставила их с бизнес-возможностями (рис. 2.5).



Рис. 2.5. Разные ограниченные контексты и разные типы общения

¹ Используется в книге Service-Oriented Architecture Томаса Эрла (Thomas Erl) (Pearson, 2005).

- Бизнес-возможность правового менеджмента отслеживают все официальные юридические документы и дела. Разные приложения, процессы и данные относятся к ограниченному контексту правового менеджмента.
- Бизнес-возможность управления клиентами ориентирована на клиентов. Вся информация о них, электронная переписка и записи телефонных звонков обрабатываются в этой системе. Все это — ограниченный контекст управления клиентами.
- Бизнес-возможность управления маркетингом передана на аутсорсинг внешней компании и представлена ограниченным контекстом управления маркетингом.

Проведя логическую границу вокруг каждого экземпляра возможности, мы ясно обозначим область каждого ограниченного контекста, использующего свой единый язык и не связанного с другими контекстами. Разделение означает отсутствие прямой связи с внутренней логикой других ограниченных контекстов. Ограниченные контексты взаимодействуют через разделяющие интерфейсы или абстракции. Для общения внутри ограниченного контекста используется единый язык. Это означает, например, что при обмене данными между системами в рамках ограниченного контекста управления клиентами для общения используется единый язык управления клиентами. В этой модели предполагается, что потребитель данных сам преобразует их в свой собственный контекст.

Шаблоны бизнес-архитектуры

На рис. 2.5 внешняя сторона предоставляет свои возможности другим ограниченным контекстам как услугу. Процесс построения бизнес-моделей платформы, передачи этих услуг другим компаниям и совместной работы над бизнес-моделями создает экосистему¹. Такая экосистема обычно состоит из нескольких участников.

Мы можем обозначить ряд моделей общения, возникающих в бизнес-экосистемах.

- *Шаг 1. Обмен данными между приложениями.* Приложения и их компоненты, использующие один ограниченный контекст и единый язык, могут обмениваться данными напрямую. Все члены команды должны понимать точки связи в ограниченном контексте.

¹ Роэль Виринга (Roel Wieringa) предоставляет фреймворк, который может помочь разделить архитектуру экосистемы (<https://oreil.ly/nOQ03>).

- *Шаг 2. Связь между предметными областями.* Взаимодействия между приложениями в разных ограниченных контекстах не могут выполняться напрямую, поскольку разные ограниченные контексты решают разные задачи и используют разные единые языки.
- *Шаг 3. Связь с внешними сторонами.* Взаимодействия с внешними системами, находящимися за границами ограниченного контекста и внутреннего доверенного окружения, должны осуществляться с применением дополнительных мер безопасности. Внешнее доверенное окружение также может считаться ограниченным контекстом, потому что внешним сторонам никогда нельзя доверять. Эта форма связи требует дополнительных мер безопасности.

В этом примере приложения из разных предметных областей напрямую взаимодействуют друг с другом. Этот шаблон также известен как *двуточечная интеграция*. В следующих разделах я представлю теорию, лежащую в основе этой и других форм связи между приложениями.

Шаблоны связи и интеграции

Прежде чем мы перейдем к решениям, я хочу кратко обсудить некоторые шаблоны и методы передачи данных. При работе с приложениями в более крупном масштабе можно использовать ряд шаблонов связи для их соединения и объединения данных. С некоторыми из них вы уже знакомы. В главе 1 я упомянул разрозненный подход к созданию монолитного приложения для потребления корпоративных данных.

Двуточечная связь

Первый шаблон — это *двуточечная связь* — прямая связь между двумя приложениями. Двуточечная связь формирует тесный контакт между приложениями. Эта форма интеграции быстро усложняется, превращаясь в полный хаос, как показано на рис. 2.6.

Чтобы проиллюстрировать состояние хаоса, давайте проведем пару вычислений и посчитаем количество соединений, используя биномиальный коэффициент. Если существует несколько приложений и каждое соединено с другими, получится $(n * (n - 1)) / 2$ двухточечных соединений. Это означает, что для управления сотней приложений понадобится примерно 5000 двухточечных соединений. Для тысячи приложений — около 500 000 двухточечных соединений.

Двухточечные соединения не могут обеспечить необходимые организациям контроль и гибкость, поскольку огромное количество каналов связи делает практически невозможным наблюдение за всеми зависимостями.



Рис. 2.6. Двухточечная связь быстро превращается в «спагетти»-архитектуру

Разрозненные хранилища

Альтернативой двухточечной связи является создание *разрозненных хранилищ*, путем объединения всех данных с логикой интеграции приложений. Преимущество разрозненных хранилищ — быстрое получение всех данных. Но в масштабе крупного предприятия использование разрозненных хранилищ для распределения данных не обеспечивает гибкости.



Я не против создания разрозненных хранилищ при разработке программного обеспечения. Модульные и распределенные конструкции увеличивают сложность и вызывают проблемы согласованности. Архитектура монолитных приложений теснее связывает их логику, что может быть полезно при наличии строгих требований к согласованности и (транзакционной) целостности. Создание разрозненных хранилищ — хороший вариант для определенных сценариев.

Разрозненность порождает несоответствия и требует огромных усилий в плане интеграции. В главе 1 мы отметили, что разрозненные хранилища данных являются одной из основных причин, по которым организации не могут воспользоваться настоящими преимуществами данных, — необходимо прилагать огромные усилия для интеграции и управления.

Звездообразная модель

Альтернативой использованию двухточечных соединений или построению разрозненных хранилищ данных является внедрение звездообразной модели. Основная идея состоит в том, чтобы создать центральный узел или уровень, соединяющий интерфейсы разных систем и приложений.

Подход к объединению данных с использованием звездообразной модели концептуально похож на разрозненное хранилище, но сильно отличается от него, так как данные в звездообразной модели не интегрируются с другими. В звездообразной архитектуре, как показано на рис. 2.7, приложения не взаимодействуют напрямую. Связь всегда идет через центральный узел. Каждое приложение должно сначала подключиться к центральному узлу и только потом обмениваться данными с другими приложениями.

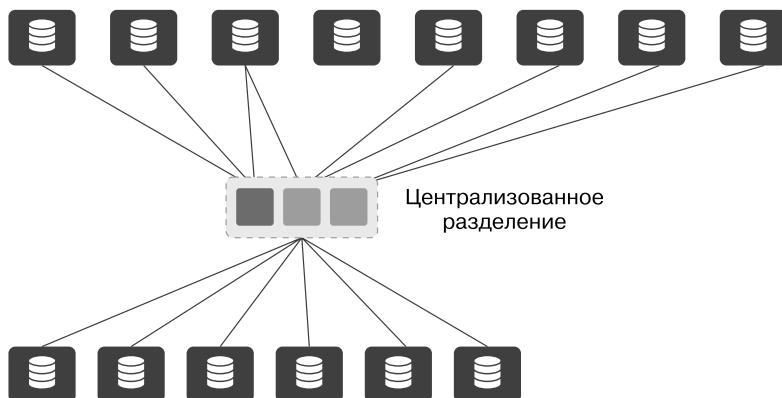


Рис. 2.7. В звездообразной модели приложения взаимодействуют не напрямую, а через центральный узел. Это существенно снижает сложность интерфейса

Сама по себе звездообразная модель не определяет выбор языка взаимодействия. Он может варьироваться от единого корпоративного языка до незавершенных форм стандартизации. Также неясно, где должна происходить интеграция: до, во время или после доставки в центральный узел. Для этого архитектурного шаблона также используются термины «центральный узел доставки данных» и «архитектура брокера данных».

Я убежден, что звездообразная модель дает значительные преимущества при переходе от централизованной и монолитной архитектуры интеграции данных к предметно-ориентированной модели распределения данных. Эту архитектуру можно рассматривать как звездообразную модель, которая также учитывает риски фрагментации и разрозненности.

Масштабируемая архитектура

Тенденции, о которых говорилось в главе 1, и теоретические основы подводят нас к новой архитектурной модели. Хотя технологии постоянно меняются, базовые концепции поставщика и потребителя данных, а также необходимость перемещать и преобразовывать данные остаются прежними. *Предприятиям нужна масштабируемая и сильно распределенная архитектура, позволяющая легко соединять поставщиков и потребителей данных и обеспечивающая гибкость, контроль и прозрачность.* Эта архитектура называется *масштабируемой*.

Новая архитектура интеграции также должна упростить работу и позволить командам эффективно сотрудничать и общаться. Она должна предоставлять шаблоны для большого количества вариантов использования и не должна заставлять компании управлять сложным двухточечным ландшафтом или тесно связанными разрозненными хранилищами данных. Компаниям нужна архитектура, позволяющая управлять зависимостями и разделять задачи в масштабе предприятия.

Масштабируемая архитектура основывается на выводах из этой главы. Работа с архитектурой предприятия для управления данными и интеграции требует стандартизации шаблонов, четкого определения принципов проектирования и принятия трудных решений. В следующих разделах я начну с основных вариантов проектирования и постепенно раскрою наиболее важные шаблоны и принципы.

Золотые источники и хранилища данных предметной области

Первый принцип масштабируемой архитектуры: разрешить распространение только золотых наборов данных из систем с золотым источником. *Золотой источник* — приложение, в котором все достоверные данные создаются в определенном ограниченном контексте. Эти уникальные данные будут связаны с *золотыми наборами данных*, которые не зависят от технологий представления, используемых для классификации данных и обеспечивающих возможность определения их владельца, классификации, контекста и т. д.

В главе 7 вы более подробно узнаете, как это работает, а пока достаточно понять, что одни и те же уникальные данные могут дублироваться и распространяться по всей архитектуре. Таким образом, классификация и идентификация уникальных данных важна при отходе от разрозненного организационного подхода. Золотые источники и золотые наборы данных обеспечивают согласованность и возможность учета при распространении данных. Мы можем сформулировать это более строго, используя следующие принципы.

- *Принцип 1: изменения происходят только в золотых источниках.* Изменения в золотом наборе данных и его элементах могут происходить только через золотой источник внутри предметной области и с одобрения владельца.
- *Принцип 2: предоставляются только золотые наборы данных.* Из системы золотого источника могут поставляться только оригинальные данные (золотые наборы данных).
- *Принцип 3: золотой набор данных – это подмножество приложения.* Золотой набор данных не может охватывать несколько приложений, потому что создается только в одном.
- *Принцип 4: золотые наборы данных и элементы регистрируются для обнаружения централизованно.* Для обеспечения прозрачности и доверия важно сделать так, чтобы метаданные золотого источника и владельца – данные о данных – были доступны централизованно с помощью инструментов и платформ. Для этого все предметные области должны централизованно регистрировать свои уникальные золотые наборы данных и элементы, а затем связывать их с данными, которые они производят и экспортят.

Теперь представим хранилища данных предметной области (domain data stores, DDS). DDS отличаются от систем с золотым источником тем, что они потребляют, интегрируют и изменяют контекст данных из других систем. Данные в этой ситуации (изображенные на рис. 2.8) возникают за пределами ограниченного контекста DDS.

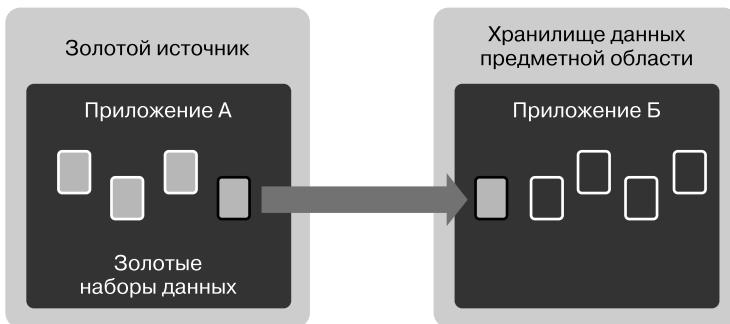


Рис. 2.8. В приложении с золотым источником управление всеми достоверными данными происходит в определенном контексте

Слева на рис. 2.8 показано приложение А, порождающее данные и управляющее ими: это система с золотым источником. Один из наборов данных используется другим приложением, приложением Б. Приложение Б в этом примере – хранилище предметных данных (DDS).

Если хранилище DDS само создает новые золотые наборы данных, оно становится новым золотым источником. В этом случае данные действительно имеют новый контекст, поэтому появляются новые факты. Также мы можем ожидать появления новых владельцев, что подводит нас к следующим трем принципам.

- *Принцип 5: новые данные приводят к появлению нового владельца.* После изменения данных и создания новых фактов ожидается, что у них будет другой владелец. Простые «синтаксические преобразования», такие как фильтрация, объединение, соединение, преобразования верхнего-нижнего регистра, переименования полей и агрегирование, не приводят к смене владельца, потому что факты остаются прежними.
- *Принцип 6: дальнейшее распространение золотых наборов данных запрещено.* Золотые наборы данных, полученные потребителями данных, не могут распространяться ими в другие предметные области без одобрения владельца или группы безопасности данных.
- *Принцип 7: не создавайте DDS без необходимости.* Избегайте создания неуправляемых копий золотых наборов данных. Они дороги в обслуживании, и их сложно контролировать. Пользователям с интеграцией данных приходится непросто, ведь они должны избегать сохранения массивов или копий данных, когда им это не нужно. Извлечение данных из приложений и их хранение в месте нахождения потребителя данных оправдано только тогда, когда необходимо сохранить вновь созданные данные.

Важно понимать, что DDS может одновременно выполнять роли потребителя и поставщика данных. В такой ситуации DDS в роли потребителя поглощает, интегрирует и преобразует данные в новые золотые наборы данных. В качестве поставщика он предоставляет вновь созданные золотые наборы данных другим потребителям. Этот процесс показан на рис. 2.9.



Рис. 2.9. Потребитель может стать поставщиком, что может привести к образованию цепочки потребления и дальнейшего распространения данных

Важно классифицировать приложение как золотой источник или как хранилище предметных данных, потому что тогда ответственность за создание золотых наборов данных станет более явной. Это также гарантирует, что данные останутся уникальными и одинаковыми на протяжении всей цепочки распределения данных.

Контракты на поставку данных и соглашения о совместном их использовании

Поставщик и потребитель данных должны создать контракт, охватывающий способы потребления, интеграции и распределения данных.

Принцип 8: распределение и потребление данных обеспечивается контрактами и соглашениями. Аспекты отчетности и управления данными, связанные с предоставлением и потреблением данных, обеспечиваются контрактом на поставку и соглашением о совместном использовании данных.

Контракт на поставку данных аналогичен контракту на обслуживание, который предоставляет гарантии как поставщикам, так и потребителям данных. Поставщик данных обещает осуществить доставку и описывает, как данные могут использоваться. Контракт гарантирует совместимость интерфейса и включает условия и соглашение об уровне обслуживания (service level agreement, SLA). Условия обслуживания описывают, как данные могут использоваться, например, только для разработки, тестирования или производства. SLA описывает качество доставки данных и интерфейса. Такое соглашение может определять долю времени безотказной работы, частоту ошибок и доступность, а также условия прекращения поддержки, стратегический план развития и номера версий (рис. 2.10).



Рис. 2.10. Поставщик данных может доставлять данные, используя сразу несколько интерфейсов и версий контрактов

Преимущество контрактов на поставку данных в том, что они дают представление о степени связанности и количестве зависимостей между приложениями. Это позволяет проводить контрактное тестирование, чтобы убедиться, что все изменения приложений и интерфейса проверены на соответствие требованиям потребителя к данным¹.

¹ Контрактное тестирование (<https://oreil.ly/mLpB>) — это методика обеспечения совместимости двух отдельных систем (например, двух микросервисов) друг с другом.

Соглашения о совместном использовании данных, в отличие от контрактов на поставку, охватывают предполагаемое использование, конфиденциальность и цель (включая ограничения). Они не зависят от интерфейса и дают представление о том, какие данные используются для конкретной цели. Потребители данных должны регистрировать и публиковать цель их потребления для различных вариантов использования, а также соглашаться не распространять данные далее. Это важно не только с точки зрения регулирования, но и потому, что дает поставщикам данных ценную информацию.

Контракты на поставку данных и соглашения о совместном использовании очень важны. Они дают представление о цепочке поставок данных и создают контракты об уровне обслуживания, элементы управления, правила качества. Кроме того, они делают использование данных прозрачными для организации. Эти контракты должны храниться в центральном репозитории — формальном строительном блоке новой архитектуры. Централизованная публикация этих контрактов позволяет поставщикам и потребителям самостоятельно решать свои проблемы с доставкой и потреблением данных, без поддержки центральной группы. Это станет особенно важным, когда мы начинаем отходить от централизованной, монолитной и разрозненной платформы данных и расширяем возможности автономных команд.

Устранение разрозненного подхода

Первый принцип проектирования масштабируемой архитектуры — не использовать разрозненные хранилища для распространения данных. Фирмы часто не могут доверять разрозненным хранилищам, потому что они требуют дополнительных этапов преобразования, что делает цепочку между поставщиком и потребителями более длинной и сложной для управления. В модели хранилища данных мы имели дело с несоответствиями при интеграции всех данных, потому что ограниченные контексты часто конфликтуют между собой. Поставщики и потребители данных в модели хранилища тесно связаны, что значительно затрудняет управление зависимостями. Организации часто не могут доверять разрозненным данным из-за проблем с качеством, исправлений и несоответствий.

Отказываясь от разрозненных хранилищ данных для распределения и используя новые, более современные формы интеграции и распределения данных (рис. 2.11), мы не позволяем командам обходить препятствия интеграции и превращать архитектуру в неуправляемую систему двухточечной интеграции. Отказ от интегрированного разрозненного хранилища заставляет искать другую модель управления зависимостями и разделения проблем в масштабе предприятия. Для сложных систем компании успешно внедряют DDD.



Рис. 2.11. Не рекомендуется распространять данные через разрозненные хранилища, такие как хранилища данных, так как ведет к повышению сложности

Предметно-ориентированное проектирование в масштабе предприятия

Самым большим изменением в управлении зависимостями и разделении задач является использование философии DDD и ограниченного контекста для установки логических границ в ландшафте приложений. Идея ограниченных контекстов, как отмечалось выше, связана с разграничением и выбором идеального размера для управления предметной областью. После идентификации ограниченного контекста его следует отделить и позволить ему взаимодействовать с другими ограниченными контекстами только через то, что я называю уровнем данных. Я объясню, как это работает, в разделе «Уровень данных как целостная картина» далее.

Использование модели предметно-ориентированного проектирования на уровне предприятия отличается от использования разрозненного подхода, где все данные централизованы в сильно интегрированной среде с корпоративными языками. В DDD каждый ограниченный контекст использует свой единый язык. При изменении ограниченного контекста меняется и единый язык. Корпоративный язык в нашей новой архитектуре становится набором нескольких единых языков.

Модель DDD не дает четких указаний о том, как определить целостность и ограниченный контекст в рамках управления данными. Поэтому я добавлю в наш список принципов проектирования новой архитектуры следующие пункты.

- *Принцип 9: данные управляются и доставляются во всех предметных областях.* Конвейер данных, качество и читаемость являются заботой предметной области. Ответственность за управление данными возлагается на людей, которые, как предполагается, лучше всех разбираются в них. Модель владения данными будет распределенной, а не централизованной.
- *Принцип 10: ограниченные контексты связаны с конкретными бизнес-возможностями.* Ограниченный контекст реализует — и, таким образом, представляет реализацию — экземпляр возможности или ее конкретную часть. Так, ограниченный контекст фокусируется на той же области решения.

- *Принцип 11: ограниченный контекст может быть связан с одним или несколькими приложениями.* Ограниченный контекст может представлять декомпозицию одного или нескольких приложений. В случае нескольких приложений ожидается, что все они будут предоставлять ценность для одного и того же (бизнес) экземпляра возможностей потока создания ценности.



Существуют разные способы разделения сложной архитектуры. Мы можем достигнуть уровня детализации и разделения с помощью предметных областей: объединить несколько приложений в пределах предметной области в ограниченный контекст и отделить от других предметных областей. Альтернативный вариант – увеличить уровень детализации и отделить приложения друг от друга. Это заставит предметные области, которые могут иметь определенную степень автономии, всегда разделять свои приложения, даже если все они находятся в одной предметной области.

- *Принцип 12: единый язык используется в ограниченном контексте.* Приложения или их компоненты, распределяющие данные в собственном ограниченном контексте, используют общий единый язык, поэтому термины и определения не противоречат друг другу. Каждый ограниченный контекст точно соответствует концептуальной модели данных.
- *Принцип 13: ограниченные контексты являются независимыми от инфраструктуры, сети и организации.* Связность – это свойство функционала приложения, процессов и данных. С точки зрения управления данными ограниченный контекст не изменяется при изменении инфраструктуры, сети или организации.
- *Принцип 14: один ограниченный контекст принадлежит одной команде.* В идеале каждый ограниченный контекст принадлежит отдельной команде гибкой разработки или DevOps-команде, потому что в единственной команде количество точек сопряжения управляемо и все они понятны ее членам.
- *Принцип 15: строгость границ.* Рамки ограниченного контекста – строгие. Каждый контекст индивидуален. Бизнес-задачи, процессы и данные, связанные друг с другом, должны оставаться вместе и поддерживаться и управляться в пределах ограниченного контекста.
- *Принцип 16: разделение при пересечении границ.* Тесная связанность допускается внутри ограниченного контекста, но при пересечении границ должны использоваться разделяющие интерфейсы.
- *Принцип 17: внутри границ допускается любая форма данных.* Внутри конкретного ограниченного контекста допускается любая форма данных при условии соблюдения явных границ.
- *Принцип 18: виртуализация и гармонизация разрешены только в ограниченном контексте.* Допускаются виртуализация и гармонизация данных, а также создание дополнительных слоев, но только в пределах ограниченного контекста.

- *Принцип 19: данные не должны доставляться через дополнительные или промежуточные системы.* Дополнительные системы, скрывающие системы с золотыми источниками, увеличивают сложность, изменяют данные и их качество, а также затрудняют отслеживание происхождения данных до источника. Использование дополнительных или промежуточных систем для передачи данных не рекомендуется.
- *Принцип 20: уровень данных не должен включать логику предметной области.* Логика предметной области и сложность интеграции должны оставаться в ограниченном контексте и не должны помещаться ни в один из компонентов интеграции. Это позволит областям сосредоточиться на предоставлении ценности, не беспокоясь о скрытой предметной логике, которая может мешать взаимодействиям с другими областями.

DDD-подход значительно увеличивает гибкость за счет ослабления связи между поставщиками и потребителями данных и устранения необходимости ждать, когда данные будут интегрированы в изолированное хранилище. Каждая бизнес-область в пределах логических границ ограниченного контекста может изменяться со своей скоростью, потому что единственны зависимости, которые она имеет, связаны с уровнем данных.

Поставщики и потребители данных, напрямую взаимодействующие со своими ограниченными контекстами, создают другие интересные результаты, которые будут рассмотрены в следующем разделе.

Преобразование в один шаг

В DDD-модели преобразования контекста выполняются напрямую, в один шаг. Данные перемещаются прямо из одного контекста в другой. Этот единственный шаг преобразования означает, что поставщик и потребитель данных больше не используют единую каноническую модель предприятия (см. глоссарий). Каждый ограниченный контекст предоставляет данные на своем языке.

Потребляющий ограниченный контекст преобразует данные непосредственно в свой собственный. Ограниченный контекст также может менять структуру данных, даже если смысл остается тем же. Избавившись от единой корпоративной модели, мы избегаем несостыковок между определениями и разными версиями истины, как это было в модели хранилища. Переход из одного контекста в другой создает максимальную ценность, потому что все имеют доступ к наиболее точным данным, близким к месту их происхождения.

Недостаток модели кроется в том, что вместо изучения единого словаря модели предприятия команды должны интерпретировать множество разных контекстов. Для этого требуются строгие принципы, определяющие, как предметные области предоставляют свои данные другим. Данные должны быть легко потребляемыми и готовыми к интеграции. Я подробнее остановлюсь на этом в главе 6.

Оптимизация данных для чтения

Перенос данных из одного контекста в другой всегда затруднен, ведь он требует знания обоих контекстов. Другая проблема — приложения в целом не оптимизированы для интенсивного потребления данных. Схемы часто сильно нормализованы, бизнес-логика скрыта внутри данных, а документации просто нет. Чтобы облегчить процесс интеграции данных, нам нужно определить некоторые принципы отображения таких данных и переноса из одного ограниченного контекста в другой.

- *Принцип 21: скрытие технических деталей приложения.* Этот принцип очень похож на принцип инверсии зависимостей (см. подраздел «Принципы объектно-ориентированного программирования» на с. 44) и влияет на выбор передаваемых данных, так как мы ожидаем, что внутренняя сложность данных будет абстрагироваться перед передачей другим потребителям данных. С точки зрения приложения это означает, что поставщики данных должны отфильтровывать любую прикладную логику, требующую определенных знаний о ней. Потребителям данных не требуется досконально знать физическую модель данных, а также восстанавливать прикладную функциональность, чтобы использовать данные. Этот принцип касается также технических данных, которые обычно присутствуют в приложениях (информация для перемещения, файлы журналов или схемы баз данных). Эти технические данные специфичны и нужны только внутри области. Поставщики данных должны отфильтровать любые не представляющие интереса данные, прежде чем экспорттировать их.
- *Принцип 22: оптимизация для интенсивного потребления данных.* Многие приложения не рассчитаны на интенсивное потребление данных. Разрешение предметным областям легко потреблять данные не означает предоставление другим областям максимально нормализованных данных с бесконечными родительско-дочерними структурами. Данные должны быть денормализованы и логически сгруппированы. Говорящие имена полей помогут потребителям найти то, что они ищут. Предполагается, что несоответствия в соглашениях об именах и форматах данных будут устраниены заранее. При таком подходе данные должны быть оптимизированы для общего потребления, чтобы сделать их как можно более пригодными для повторного использования и удовлетворить потенциальные нужды всех потребителей данных. Это связано с увеличением соотношения чтения и записи, как описано в главе 1.
- *Принцип 23: единый язык — это язык общения.* Каждый ограниченный контекст, действующий в роли поставщика, должен предоставлять свои данные, используя собственный единый язык. Это означает, что поставщики данных не должны включать бизнес-логику из других предметных областей в свою.
- *Принцип 24: интерфейсы должны иметь определенный уровень зрелости и устойчивости.* Этот принцип заключается в абстрагировании скорости изменений, если диапазоны областей часто меняются, например, ожидается

переход к более устойчивому диапазону. Схема также должна обеспечивать устойчивый уровень совместимости.

- *Принцип 25: данные должны быть согласованы по всем шаблонам.* Этот принцип требует согласованности в представлении данных в разных шаблонах. Например, поле «адрес клиента» должно быть согласовано во всех шаблонах, даже если одни и те же данные представлены несколько раз.

Подход с извлечением описательных данных, удобных для пользователя, существенно отличается от многих реализаций озер данных. В озере данных (см. подраздел «Озеро данных» на с. 35) обычно данные извлекаются напрямую из источника в виде точной копии. Данные и интерфейсы тесно связаны с базовыми исходными системами, поэтому любое изменение исходной системы немедленно нарушит работу конвейера. По этой причине предприятия испытывают трудности при внедрении продвинутой аналитики.

Завершая перечисление основных принципов, можем заключить, что поставщики данных должны предоставлять их удобным, понятным и логичным способом. Следует избегать повторной обработки необработанных данных. Данные должны быть представлены в виде абстрактной версии «логической бизнес-модели», а не чисто физической модели данных из приложения. В главе 6 я подробнее расскажу, что такое хорошее представление данных.

Перечисленные принципы не требуют строгого соблюдения. При проведении исследований и экспериментов интерфейс может быть более изменчивым или менее оптимизированным для прямого потребления. В промышленном окружении, напротив, следует строго придерживаться этих принципов. Вы даже можете смешивать подходы, комбинируя их внутри системы: например, один набор данных может быть доставляться в строгом соответствии с принципами, а другой (возможно, что-то менее интересное) — с использованием более свободного подхода.

Принципы важны, потому что новая архитектура по-разному облегчает распределение и интеграцию данных. Если вы хотите перейти к управляемым данным и стимулировать их потребление, то прежде всего должны пересмотреть способ предоставления данных во всех предметных областях. Области должны придерживаться перечисленных принципов и применять их ко всем моделям взаимодействий, о которых мы поговорим в следующих главах.

Уровень данных как целостная картина

Мы подошли к самой интересной части новой архитектуры, а именно к единому подходу к распределению и интеграции данных между областями. Как вы узнали выше, данные всегда должны преобразовываться, поэтому нужна новая

модель, упрощающая этот процесс, снижающая сложность двухточечного соединения и позволяющая избежать разрозненного подхода. Это можно сделать через *уровень данных* — строго управляемую и защищенную архитектуру, помогающую предметным областям распределять данные с использованием множества шаблонов.

Прежде всего, уровень данных позволяет предметным областям легко расширяться, чтобы обеспечить потребление и распространение данных. Он использует базовую инфраструктуру и платформы для распределения данных, чтобы гарантировать соблюдение всех требований к управлению. Как только предметная область начинает экспортirовать данные, остальные стороны могут начать потреблять их, следуя принципам *оптимизации данных для потребления*. На рис. 2.12 показана эта идея в общем виде.

Подход с использованием уровня данных похож на звездообразную модель, ведь он гарантирует прохождение всех данных через один и тот же центральный узел — уровень данных. Создав общую картину всех потоков связи, можно видеть, какие приложения подключены к уровню данных, какие данные участвуют в обмене, как данные должны распределяться, каково качество данных, какую роль — потребителя или поставщика — играет та или иная область, из какого контекста она возникла и т. д. Другими словами, все возможности управления данными будут развернуты на этом уровне.

На уровне ниже этот подход работает иначе. Цифровые платформы и возможности (самообслуживания) объединены в новые слои (архитектурные шаблоны), чтобы учесть масштабы объемов, скорости и разнообразие. С концептуальной точки зрения виден один абстрактный и упрощенный уровень данных, но внутри мы имеем три архитектуры, тесно взаимодействующие друг с другом, чтобы сохранять¹, направлять, преобразовывать, распределять и копировать данные в различные конечные точки. Многообразие шаблонов нужно для разных сценариев использования: ни одна платформа и ни одно решение интеграции данных не способны удовлетворить все требования интеграции. Эта идея проиллюстрирована на рис. 2.13.

В следующем разделе обсуждаются три архитектуры, полностью управляемые метаданными, помогающие определить службы данных, которые можно повторно использовать в других приложениях-потребителях. Они также дают представление о происхождении распределения, правилах потребления, качестве данных, значении и т. д. Предоставляя эти метаданные через API, поставщики и потребители могут улучшить эффективность взаимодействий с различными архитектурами и автоматизировать свои конвейеры данных. Все это будет подробно описываться в оставшейся части книги.

¹ Ожидается, что информация на уровне данных будет только храниться и едва ли — изменяться.

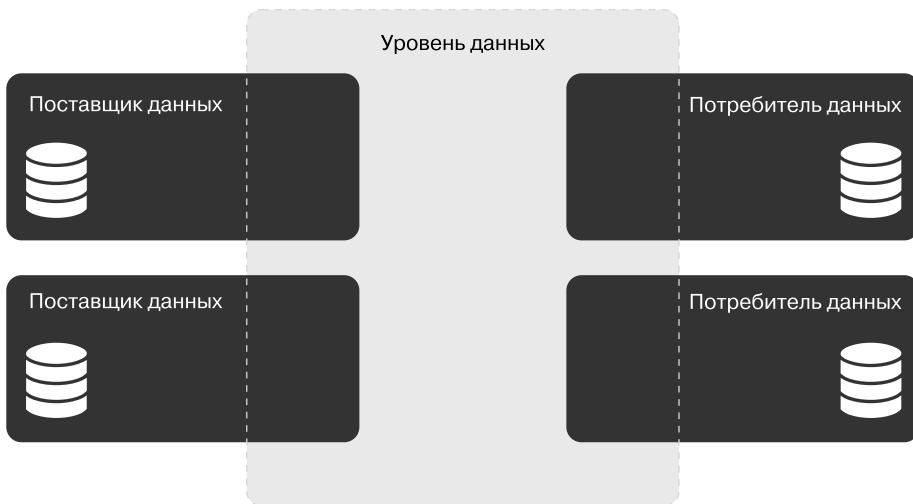


Рис. 2.12. Общее представление архитектуры. Области расширяются, образуя уровень данных и обеспечивая потребление и распространение данных в масштабах всего предприятия

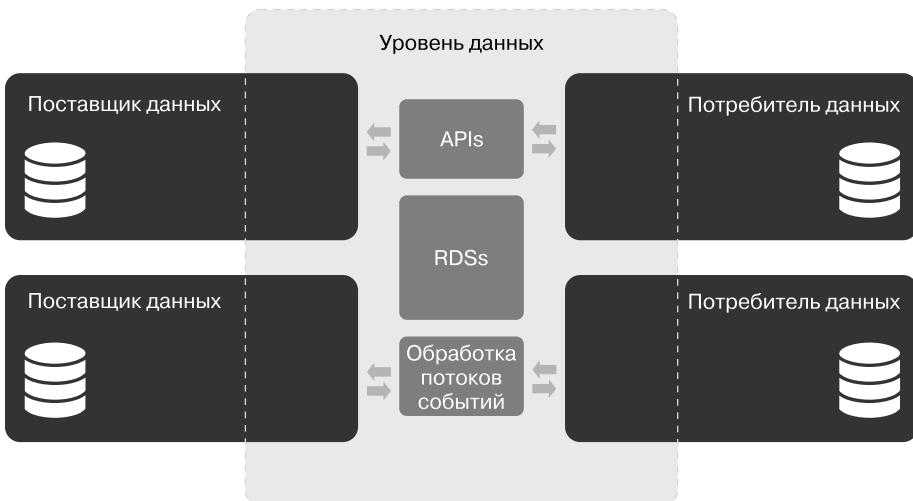


Рис. 2.13. Более детальный взгляд на архитектуру, работающую с тремя другими архитектурами с различными возможностями

Многоуровневый подход много значит для управления сложностью взаимодействия, ведь архитектуры стандартизированы в трех схемах связи. Специалистам сложно работать с большим количеством разных форматов, поэтому на помощь приходят архитектуры. Они дают возможность использовать множество различных моделей и форматов данных, обеспечивая гибкость, которая нужна

многим предприятиям. В зависимости от варианта использования и требований к интеграции уровень данных предлагает на выбор три типовые архитектуры распределения и интеграции данных.

Архитектура хранилища данных только для чтения

Первая архитектура — хранилище данных только для чтения (read-only data stores, RDS) — позволяет поставщикам открывать доступ к данным, организуя разнообразные *хранилища данных только для чтения*. Это облегчает воспроизведение данных только для чтения (то есть неизменяемых). Хранилища RDS легко масштабируются и упрощают сложную обработку данных, такую как преобразование или очистка. Их можно использовать и для операционных процессов. А еще они упрощают реализацию различных подходов к доставке, таких как процессы ETL, за счет использования интуитивно понятных и стандартных возможностей и фреймворков, тесно взаимодействующих с RDS. Кроме того, исторические данные, потерявшие актуальность с точки зрения системы-источника, тоже могут перемещаться и сохраняться в RDS. Сохранение исторических данных позволяет воспроизводить варианты использования, выполнять специальный анализ, отлаживать и выяснять, какие данные доступны и были созданы в прошлом. Архитектура RDS будет подробно рассмотрена в главе 3.

Архитектура API

Архитектура API применяется для подключения сервисов и распределения небольших объемов данных для использования в реальном времени и с малой задержкой. Эта архитектура упрощает операции записи, обновления и удаления, тогда как архитектура RDS поддерживает только операции чтения. Шаблоны API и эволюция сервис-ориентированной архитектуры и микросервисов будут подробно рассмотрены в главе 4.



Я использую термины «сервис» и «API» как синонимы, но между ними есть некоторые тонкие различия. Веб-сервисы обмениваются данными между приложениями по сети; API могут использовать любой тип связи. Например, вы можете вызвать Microsoft Windows API, чтобы нарисовать указатель на экране.

Потоковая архитектура

Эта архитектура ориентирована на потоковую передачу больших объемов событий и сообщений в реальном времени. Потоковая передача отличается от API тем, что она асинхронна, отличается высокой пропускной способностью и может использоваться для копирования состояния приложения. Шаблон потоковой передачи будет подробно рассмотрен в главе 5.

Метаданные и целевая операционная модель

Все три архитектуры используют метаданные (рис. 2.14) гораздо интенсивнее, чем многие другие. Эти метаданные применяются для внутренней маршрутизации и распределения, а также для определения смыслового значения и проверки целостности данных. Эта информация будет раскрыта в главах 6 и 10, а пока рассмотрим роль метаданных в распределенной архитектуре с использованием различных целевых операционных моделей.

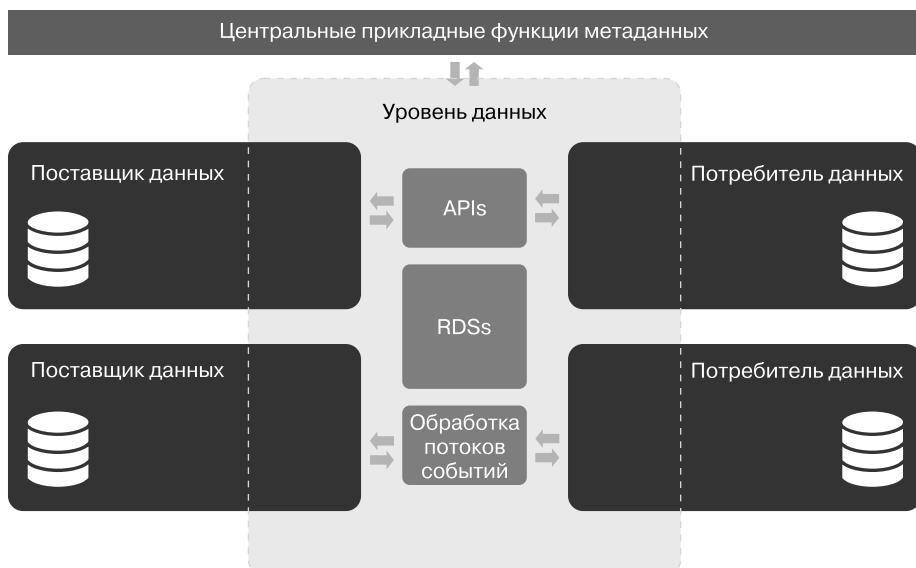


Рис. 2.14. Общее представление трех различных архитектур и метаданных

Метаданные позволяют упростить интегрированную реализацию масштабируемой архитектуры. Целевую операционную модель можно изменить и дать возможность предметным областям или командам самостоятельно создавать и использовать части архитектуры либо выполнять интеграцию. Другими словами, модель можно изменить так, чтобы дать предметным областям гибкость в выборе, создании и запуске, например, любой базы данных, которая ведет себя как RDS. Области в этой модели должны сами подключать свои решения к функциям центрального управления. Управление метаданными (право собственности, информация о схеме, соглашение о совместном использовании и т. д.) всегда должно оставаться централизованным.

Можно даже применять гибридную целевую операционную модель: например, центральная группа может предлагать наиболее часто используемые

возможности, а другие области или группы могут применять некоторые специальные возможности. Имея метаданные, вы получаете контроль над данными и их понимание.

Централизованное хранение метаданных позволяет создать схему всего приложения и архитектуры. Приложения и данные связаны, а поскольку вы знаете, какие данные проходят через уровень данных, то можете визуализировать их происхождение. Можно даже применить машинное обучение для оптимизации архитектуры, отыскивая подходящие структуры областей. Если разные области интенсивно взаимодействуют друг с другом, то, скорее всего, рамки ограниченного контекста размещены неверно. Интенсивный обмен данными между службами обычно означает, что они фактически должны быть одним целым. Как видите, метаданные играют ключевую роль в общей архитектуре. В главе 10 метаданные рассматриваются подробно.

Итоги главы

Давайте кратко перечислим все, что обсудили к настоящему моменту. Избавляясь от централизованных монолитов, вы позволяете предметным областям или командам независимо изменять данные и обмениваться ими в интегрированной модели. Каждая область владеет частью общей архитектуры. Интеграция и гармонизация данных выполняются областями, но наличие уровня данных позволяет стандартизировать архитектуры, архитектурные блоки, инфраструктуру для быстрого обмена данными и интеграции. Этот уровень находится под централизованным управлением безопасностью и контролем и реализуется на основе четких принципов организации легко потребляемых данных. Хотя ландшафт разбит на мелкие части, данные по-прежнему остаются доступными. Наконец, разрешение областям выполнять конкретные бизнес-задачи с большим разнообразием шаблонов интеграции позволяет использовать широкий спектр разнообразия приложений.

Логически группируя приложения в ограниченном контексте по уровням бизнес-возможностей и потоков ценностей, вы гарантируете, что приложения, решая однотипные задачи, находятся под единым управлением и не имеют прямых зависимостей от приложений, управляемых другими бизнес-подразделениями. Такой подход обеспечивает ясную и гибкую организацию. Команды могут полностью сосредоточиться на предоставлении ценности отдельных бизнес-возможностей. Каждая область, логически ограниченная контекстом, поддерживает определенную часть общей архитектуры, включая авторитетные уникальные наборы данных.

Подключая свои приложения к уровню данных и применяя единое преобразование контекста, вы гарантируете прохождение всех данных через один и тот же логический уровень, что обеспечивает максимальную прозрачность и скорость потребления данных. Все данные будут направляться, распределяться и извлекаться из одного логического места. Принцип ответственности за предоставление качественных данных в удобном для пользователя виде обеспечивает разделение и облегчает повторное использование этих данных. Добавление метаданных в уровень данных позволяет глубже понять все области управления данными. Уровень данных обеспечивает это понимание и позволяет контролировать подключение приложений и обмен такими данными.

Достичь такого уровня зрелости, когда интеграция данных становится простой и управляемой, сложно. Для этого необходимы осознанный выбор и вложения в централизованные возможности: ни одно интеграционное решение или платформа не могут удовлетворить все варианты использования. Для создания этих возможностей требуются стандартизация, применение передовых практик и опыт поддержки групп разработчиков и проектов. Для наращивания централизованных возможностей нужно, чтобы отдельные группы отказались от своих полномочий принимать решения о шаблонах интеграции и инструментах. Это может вызвать сопротивление и поставить вас перед выбором. Кроме того, такая модернизация ландшафта данных требует прагматичного подхода, ведь отойти от тесно связанного представления сложно. Начав с малого, с простых потоков данных, и постепенно увеличивая масштаб, вы продемонстрируете преимущества для предметных областей и пользователей — и они сами захотят внести свой вклад в новую архитектуру, которая позволит организации превзойти конкурентов. Со временем архитектуры интеграции могут быть расширены за счет добавления новых источников, шаблонов интеграции и возможностей, что способно улучшить качество данных и понимание в масштабах всего предприятия.

Новая архитектура интеграции должна охватывать все предприятие. Без этого потенциал архитектуры не может быть полностью реализован. Команды, которые сами развертывают сервисы данных и просто заменяют разрозненные хранилища другими сервисами, подвергаются риску «разрастания данных»¹. Повторное использование, согласованность и качество данных являются важными аспектами новой архитектуры. Для увеличения масштабов необходимы последовательная коммуникация, приверженность принципам и уверенное управление.

¹ Под разрастанием данных подразумевается огромное количество и разнообразие данных, производимых предприятиями каждый день.

ГЛАВА 3

Управление большими объемами данных: архитектура хранилищ данных только для чтения

В этой главе будет рассмотрена архитектура RDS. Она предназначена для активного чтения. Вы познакомитесь со схемой разделения ответственности команд и запросов (Command and Query Responsibility Segregation, CQRS), мы также добавим новые принципы в нашу общую архитектуру. Дальше мы рассмотрим хранилища данных, доступные только для чтения (read-only data stores, RDS): что они собой представляют, как проектируются и какими возможностями обычно должны обладать, а также роль метаданных. К концу главы вы поймете, как RDS могут помочь сделать огромные объемы данных доступными для потребителей.

Знакомство с архитектурой RDS

С тех пор как хранилища данных стали обычным товаром, многое изменилось. Стали популярными распределенные системы, объем данных увеличился и разнообразился, появились новые виды баз данных. С появлением облачных технологий вычислительные ресурсы и ресурсы хранения разделились, чтобы повысить масштабируемость и эластичность. Объедините эти тенденции с проблемами, обсуждаемыми в главе 1, и принципами разделения, о которых вы узнали в главе 2, и сразу поймете важность изменения способа распределения и совместного использования больших объемов данных.

Архитектура RDS – это первая архитектура распределения и интеграции данных, которая представляет наибольший интерес. Это краеугольный камень новой масштабируемой архитектуры. Она предназначена для активного чтения и обеспечивает управляемый и безопасный доступ к объединенным данным для самых разных рабочих нагрузок. Архитектура может быть спроектирована

путем сочетания различных технологий, упрощающих разные варианты использования, такие как машинное обучение или обмен данными с третьими сторонами.

RDS в масштабируемой архитектуре имеют техническое сходство с хранилищами данных. Например, традиционная автономная пакетная обработка — это метод распределения данных, хорошо знакомый инженерам хранилищ данных. С другой стороны, RDS значительно отличаются от традиционных хранилищ: они используют язык предметной области, являются неизменяемыми, могут хранить гораздо больше данных, имеют другие шаблоны приема и оптимизированы для процессов с интенсивным использованием данных, таких как продвинутая аналитика и бизнес-аналитика.

RDS предназначены для обслуживания и предоставления больших объемов данных потребителям и позволяют выполнять запросы к самим себе, а не к лежащей в основе базе данных. Они могут надежно распределять данные из одного окружения в другое, не создавая дополнительной сложности и не теряя контроля. Их также можно использовать для непрерывного мониторинга качества данных и хранения исторических данных, что обеспечивает эффективное управление их жизненным циклом.

Разделение ответственности команд и запросов

Прежде чем углубиться в тему шаблонов, передовых практик и принципов, я хочу напомнить вам некоторые факты. В главе 1 вы узнали о технических проблемах, связанных с ограничением передачи данных, проектированием транзакционных систем и обработкой быстро растущего потребления данных. Извлечение больших объемов данных из высоконагруженной операционной системы может быть рискованным, потому что системы при слишком большой нагрузке могут стать непредсказуемыми, недоступными или, что еще хуже, выйти из строя. Вот почему разумно создать копию запрошенных данных, доступную только для чтения, и затем сделать ее доступной для потребления.

Что такое CQRS

CQRS (<https://oreil.ly/qmeLE>) — это шаблон проектирования приложений, основанный на создании копии данных для активного чтения.

Операционные команды и аналитические запросы (часто называемые *записью и чтением*) выполняют очень разные функции и в шаблоне CQRS

обрабатываются отдельно (как показано на рис. 3.1). Подсчитав объем операций, выполняемых в нагруженных системах, вы обнаружите, что команды используют в основном вычислительные ресурсы. Это логично, потому что для успешной (то есть надежной) записи, обновления или удаления данных обычно требуется выполнить ряд шагов.

1. Проверить доступный объем памяти.
2. Выделить дополнительную память для записи.
3. Получить метаданные таблицы/столбца (типы, ограничения, значения по умолчанию и т. д.).
4. Найти запись (при обновлении или удалении).
5. Заблокировать таблицу или запись.
6. Записать новые данные.
7. Проверить добавленные значения (например, на уникальность, корректность и т. д.).
8. Зафиксировать данные.
9. Снять блокировку.
10. Обновить индексы.

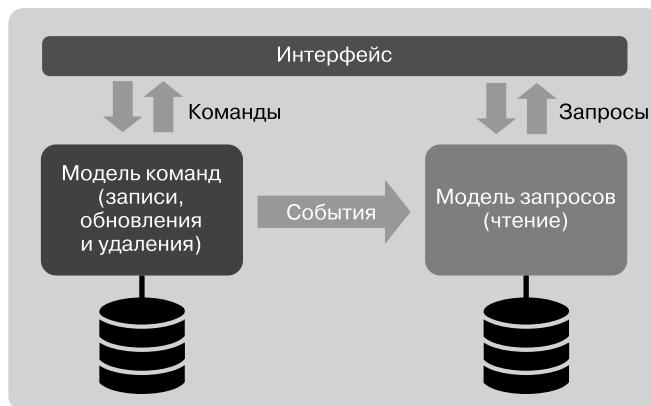


Рис. 3.1. CQRS разделяет запросы и команды, используя две разные модели данных: модель команд для транзакций и модель запросов для операций чтения

По сравнению с записью чтение из базы данных требует меньшего количества вычислительных ресурсов, потому что при чтении выполняется меньшее количество задач. Для оптимизации шаблон CQRS отделяет операции записи

(команды) от операций чтения (запросы) с помощью двух моделей. После разделения они должны быть синхронизированы, это обычно делается путем публикации событий с изменениями, что показано стрелкой «События» на рис. 3.1.

Преимущество CQRS в том, что вы не привязаны к одному типу базы данных для записи и чтения¹. Вы можете оставить базу данных для записи объективно сложной, а базу данных для чтения оптимизировать для увеличения производительности операций чтения. Если у вас разные требования для разных вариантов использования, то можно даже создать несколько БД для чтения, каждая из которых будет иметь оптимизированную модель чтения для конкретного варианта использования. Различные требования позволяют иметь и разные модели согласованности между RDS, даже при использовании одного и того хранилища данных для записи.

Еще одно преимущество CQRS заключается в отсутствии необходимости одновременно масштабировать операции чтения и записи. Когда у вас заканчиваются ресурсы, вы можете масштабировать какую-то из операций. Последним важным преимуществом отсутствия единой модели обслуживания операций записи и чтения является технологическая гибкость. Если нужна высокая скорость чтения или другие структуры данных, то можно выбрать для этого другую технологию, соблюдая свойства базы данных ACID (atomicity, consistency, isolation, durability — «атомарность, согласованность, изоляция, долговечность»).



Шаблон CQRS тесно связан с материализованными представлениями². Материализованное представление внутри БД — это физическая копия данных, которая постоянно синхронизируется с базовыми таблицами. Материализованные представления часто используются для оптимизации производительности. Вместо того чтобы запрашивать данные из базовых таблиц, запрос выполняют к предварительно вычисленному и оптимизированному подмножеству, которое находится внутри материализованного представления. Это разделение базовых таблиц (записи) и материализованного подмножества (чтения) — такое же разделение, которое наблюдается в CQRS, с той небольшой разницей, что оба набора данных находятся в одной БД, а не в двух разных.

Хотя CQRS — это шаблон проектирования программной инженерии, способный упростить процесс разработки конкретной (и, возможно, более крупной)

¹ Недостатком CQRS является усложнение задачи — необходимо синхронизировать две модели добавлением дополнительного слоя. Это также может вызвать дополнительную задержку.

² TechDifferences (<https://oreil.ly/PAoPD>) представляет обзор различий между простыми и материализованными представлениями в базе данных.

системы, он во многом послужил основой для замысла и концепции архитектуры RDS. *Новая модель архитектуры заключается в том, что всякий раз, когда другим приложениям нужно активно читать данные, для каждого из них создается минимум один RDS.* Такой подход упрощает разработку и реализацию архитектуры, а также улучшает масштабируемость для интенсивного потребления данных.

CQRS в масштабе

Использованием реплик в качестве источника для чтения данных уже никого не удивить. Фактически каждое чтение из реплики, копии, хранилища или озера данных можно рассматривать как форму CQRS. Разделение команд и запросов между системами оперативной обработки транзакций (online transaction processing, OLTP) (которые обычно пользуются и управляют приложениями, ориентированными на транзакции) и хранилищем операционных данных (operational data store, ODS, используется для оперативной отчетности) аналогично. ODS в этом примере содержит копию данных из системы OLTP. Все эти шаблоны следуют философии CQRS, которая заключается в создании выделенных баз данных для чтения на основе оперативной базы данных.



Мартин Клеппманн (Martin Kleppmann) использует «шаблон выворачивания базы данных наизнанку» (<https://oreil.ly/YToL6>), еще одно воплощение CQRS, в котором упор делается на сбор фактов посредством потоковой передачи событий. Этот шаблон пересекается с RDS, и мы рассмотрим его в главе 5.

RDS создаются по тому же шаблону CQRS. Они занимают позицию копии данных и находятся на уровне данных между поставщиком и потребителем. Они строго управляются и наследуют контекст от предметных областей. На высоком уровне это означает, что, когда поставщики и потребители предполагают обмениваться большими объемами данных, сначала создается хранилище данных только для чтения (рис. 3.2).

Обратите внимание, что хранилище RDS изображено слева, рядом с поставщиком данных, а этап преобразования — рядом с потребителем. Такое расположение подчеркивает единство подхода к распределению и интеграции данных. Вместо интеграции с использованием корпоративной или единой унифицированной модели данных проект изменен так, чтобы предоставить БД с данными для чтения всем приложениям-потребителям. Эти БД не предназначены для каких-либо действий. Их цель — предоставление данных для интенсивного чтения. Поэтому природа этих данных отличается от природы оперативных данных: они

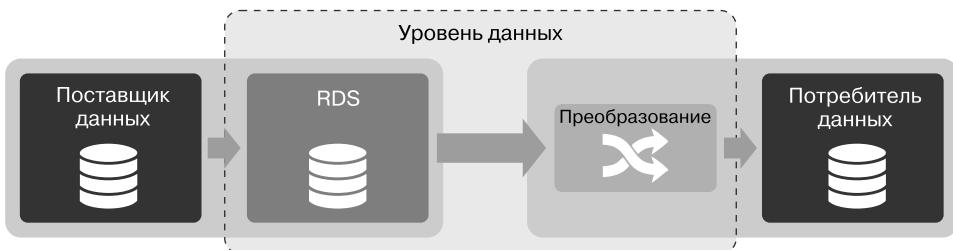


Рис. 3.2. Хранилище данных только для чтения наследует контекст области от поставщика данных и, следовательно, считается частью области, предоставляющей данные. Когда данные извлекаются, контекст преобразуется в язык потребителя, поэтому такой шаг находится в компетенции потребителя

имеют больший объем, менее изменчивы и основаны на фактах, но, чтобы этот механизм работал, необходимо соблюдать ряд принципов.

- RDS являются *неизменяемыми базами данных и доступны только для чтения*. Почему неизменяемыми? Потому что из них мы можем восстанавливать одни и те же данные снова и снова. При использовании БД, доступной только для чтения, нельзя получить разные версии истины для одних и тех же предметных данных. Согласно этому принципу RDS не создают новых данных. «Истину» можно доработать только в приложении с золотым источником.
- В RDS разрешается доставлять лишь оригинальные надежные золотые наборы данных. Использование уникальных данных гарантирует существование одного источника истины как для потребителей данных, так и для средств управления ими. Эти два элемента соответствуют нашему обсуждению в подразделе «Золотые источники и хранилища данных предметной области» на с. 60.
- Структура RDS определяется контектом поставщика данных. Следуя идеи предметно-ориентированного проектирования, модели наследуются от поставщика данных (ограниченный контекст) и основываются на реальных задачах. Модели предприятия не используются.
- RDS предназначены для приложения. Это означает, что они изолированы и не могут использоваться напрямую или иметь созависимости с другими приложениями. Этот принцип, так же, как предыдущий и два следующих, тоже основывается на обсуждении в подразделе «Предметно-ориентированное проектирование в масштабе предприятия» на с. 65, где подчеркивалась важность строгости границ.
- Никакая бизнес-логика или логика интеграции любых потребителей данных не может быть помещена внутрь RDS, принадлежащих поставщику данных.

Бизнес-логика, зависящая от предметной области, должна оставаться в границах области и не выходить за них. RDS считается частью поставщика данных, но также является и границей. Разделяя эти проблемы, мы стремимся к максимальной гибкости.

- Потребителям данных *не разрешается создавать новые данные* непосредственно в RDS. Потребители могут стать поставщиками, но сначала они должны экспортить данные с платформы, где размещены RDS. На общей платформе потребители должны сначала передать их на платформу потребления, прежде чем передать их обратно в новую RDS. В противном случае вы рискуете получить бесконечные слои и массивные данные, постоянно разрастающиеся в объеме.
- Сложность данных должна абстрагироваться, чтобы скрыть их основной источник. Этот и следующий принцип соответствуют обсуждению из подраздела «Оптимизация данных для чтения» на с. 68.
- RDS должны оптимизироваться для потребления данных и могут содержать подмножество данных приложений. Это будет подробно объяснено в главе 6, но, хотя контекст должен оставаться тем же, структуру данных допускается изменять.
- Каждое приложение или система получает как минимум один экземпляр RDS. Первый расположен рядом с золотым источником, например, в том же месте в локальной или облачной инфраструктуре.
- Поставщик данных несет ответственность за качество, структуру и доставку данных в RDS.

После доставки RDS данные можно использовать для потребления. Как обсуждалось в главе 1, каждое приложение предъявляет особые и уникальные требования. Поэтому потребитель нуждается в этапе преобразования из контекста в контекст. Новая архитектура должна способствовать этому, давая необходимое понимание всех преобразований и перемещений данных. По этой причине этап преобразования также попадает в круг обязанностей уровня данных, как показано на рис. 3.3.

Вносимое изменение делит модель приложения на модели обновления и чтения, которые называются моделями команд и запросов. Причина деления модели приложения на две части — масштабирование и подготовка к современному миру интенсивного использования данных. Когда данные преобразуются в другой контекст, создаются новые данные и ожидается смена владельца. По этой причине золотые источники, RDS и приложения-потребители, такие как DDS, четко разделены.

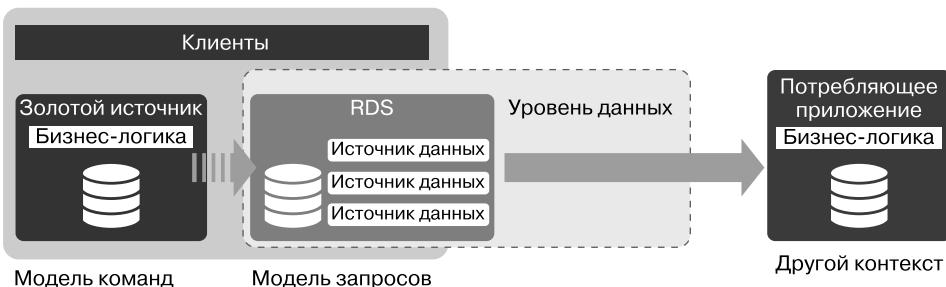


Рис. 3.3. В шаблоне CQRS RDS играют роль модели запроса

Как видите, этот подход отличается от многих реализаций хранилищ или озер данных. Хотя предлагаются централизованные платформы и возможности, данные остаются контекстом предметной области и не могут преобразовываться в модель предприятия или другой формат. Необработанные данные также не являются точной копией исходных систем. Данные оптимизированы для простого и многократного использования. Многие из этих аспектов будут подробно объяснены в главе 6.

Создание RDS для потребителей данных имеет еще одно преимущество — их можно использовать в операционной сфере. Они позволяют потребителям читать данные и одновременно приносят пользу и поставщикам. Сюда входят и сценарии, требующие большого количества операций чтения данных в рабочем контексте. Имейте в виду, что в короткие периоды чтение может вернуть рас согласованные данные из-за наличия некоторой задержки между обновлением на стороне команды и обновлением на стороне запроса (RDS).

В масштабируемой архитектуре несколько RDS могут объединяться и дублировать данные друг друга. Например, шаблон одного потребителя может использовать и комбинировать данные из нескольких RDS. Другой шаблон может дублировать RDS в другие RDS.

Полная разбивка с различными шаблонами показана на рис. 3.4.

Прочитав это, вы можете задаться вопросом, как устроены эти RDS. Монолитная ли это платформа или каждый владелец приложения должен обслуживать и размещать собственное хранилище данных только для чтения? Какие возможности преобразования и интеграции необходимы для удовлетворения различных нужд потребителей? Поддерживается ли эта архитектура метаданными и возможностями самообслуживания? Как различные компоненты этой архитектуры взаимодействуют между собой? Все эти вопросы будут рассмотрены в следующих разделах.

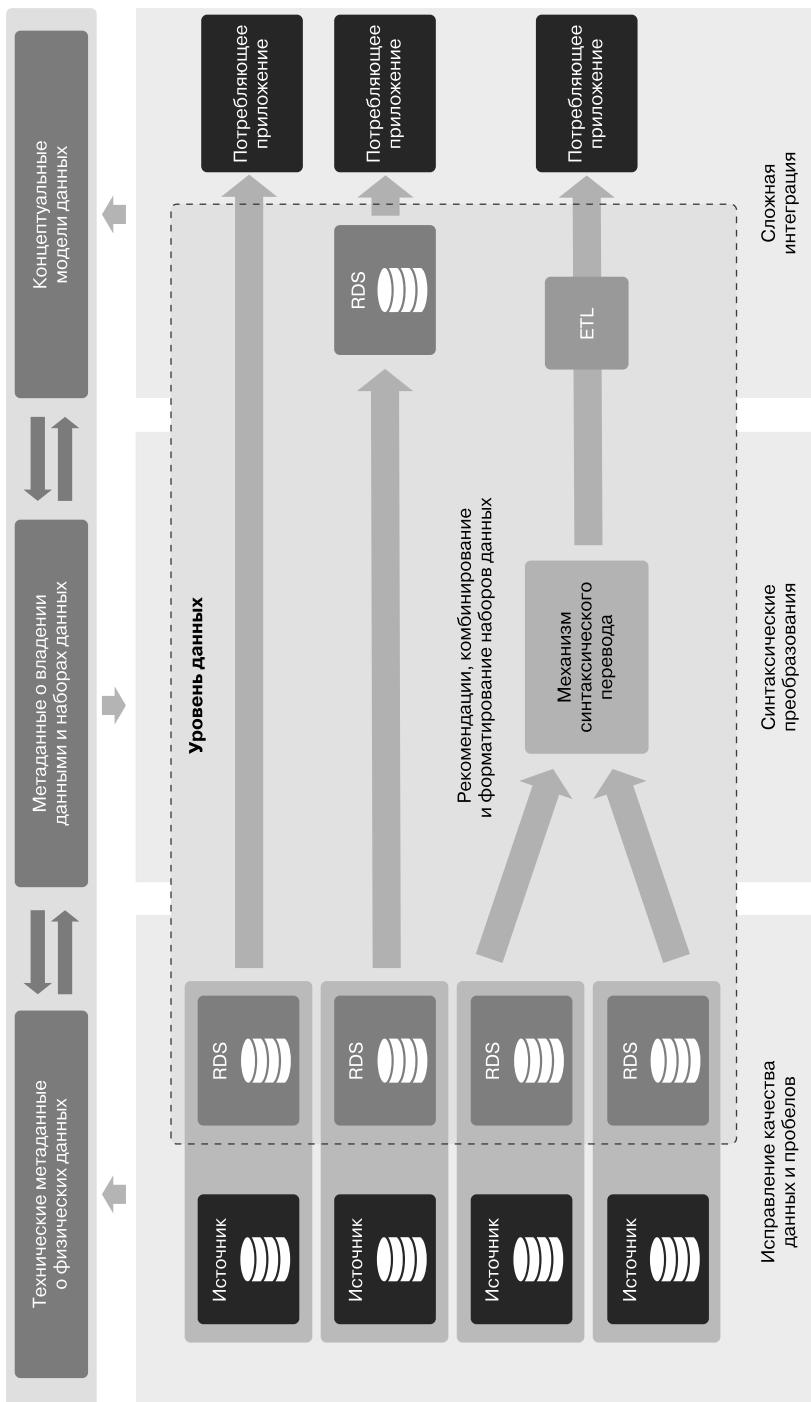


Рис. 3.4. RDS действуют как уровень абстракции для потребителей данных. Способ передачи данных в экосистеме — это скорее техническая деталь. Самое главное, что данные становятся доступными

Службы и компоненты хранилища данных только для чтения

Чтобы сделать наш ландшафт проще, нужно найти новый баланс. Интегрированный подход, позволяющий каждой предметной области размещать свое хранилище RDS и использовать свой технологический стек, приведет к увеличению количества продуктов, протоколов интерфейса, информации в метаданных и т. д. Это значительно усложнит управление данными и контроль, ведь в каждой области должны четко реализовываться основные дисциплины (архивирование, преемственность, обеспечение качества и безопасности данных). Разработка всего этого в самих предметных областях будет сложной задачей и приведет к фрагментированности и разрозненности.

Другой подход — централизация. Я говорю не о создании единого разрозненного хранилища, а о нескольких самообслуживаемых платформах, использующих одну инфраструктуру. Общие платформы и абстрагируемая инфраструктура сокращают затраты, уменьшают количество интерфейсов и улучшают контроль. В идеале несколько RDS следует размещать на одной физической платформе, но они должны быть изолированы (как показано на рис. 3.5). В зависимости от нужд потребителей и форматов наборов данных существуют варианты, оптимизированные для чтения (я вернусь к этому позже в подразделе «Варианты проектирования» на с. 101). RDS могут иметь разные формы: корзины, папки, высокопроизводительные базы данных, виртуализированные экземпляры и т. д. Выбор формы зависит от выбранной базовой технологии и требований сценария использования.

Ключ к проектированию общих платформ и RDS — предоставление набора центральных базовых компонентов и компонентов, не зависящих от предметной области. Принятие различных конструктивных решений тоже важно, чтобы отделить повторяющиеся задачи от предметных областей и упростить управление данными сложным. В следующих разделах я выделю самые важные моменты.

Метаданные

Первое, что нужно учитывать при проектировании, — это создание общих платформ, соответствующих таким критериям управления корпоративными данными, как сбор и организация важных метаданных. После создания общих платформ вы сможете автоматически извлекать показатели производительности интерфейсов, качества данных, применяемой к данным логики происхождения и преобразования, схемы данных в источниках, используемых интерфейсами, и т. д. На рис. 3.6 показана типовая модель.

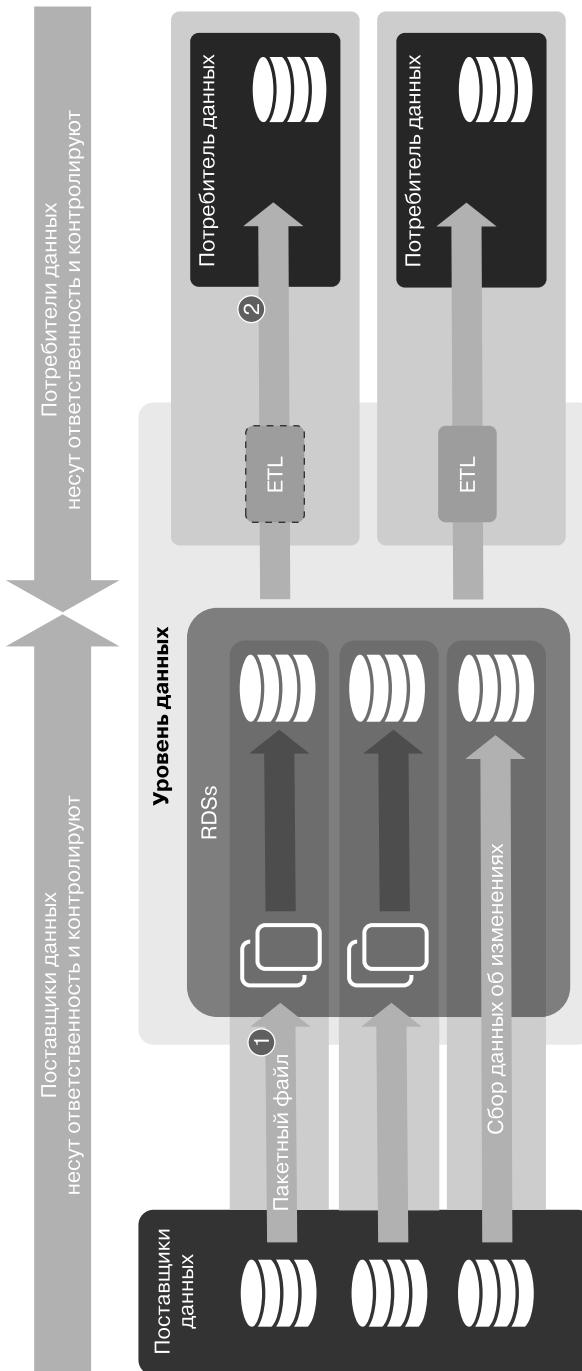


Рис. 3.5. При использовании общей инфраструктуры различные RDS изолированы и не используются напрямую между предметными областями. Каждый RDS и отчетность остаются за поставщиком данных. Поставщик данных и область имеют свой собственный экземпляр RDS и свой конвейер. Поставщик берет на себя ответственность за качество и целостность данных

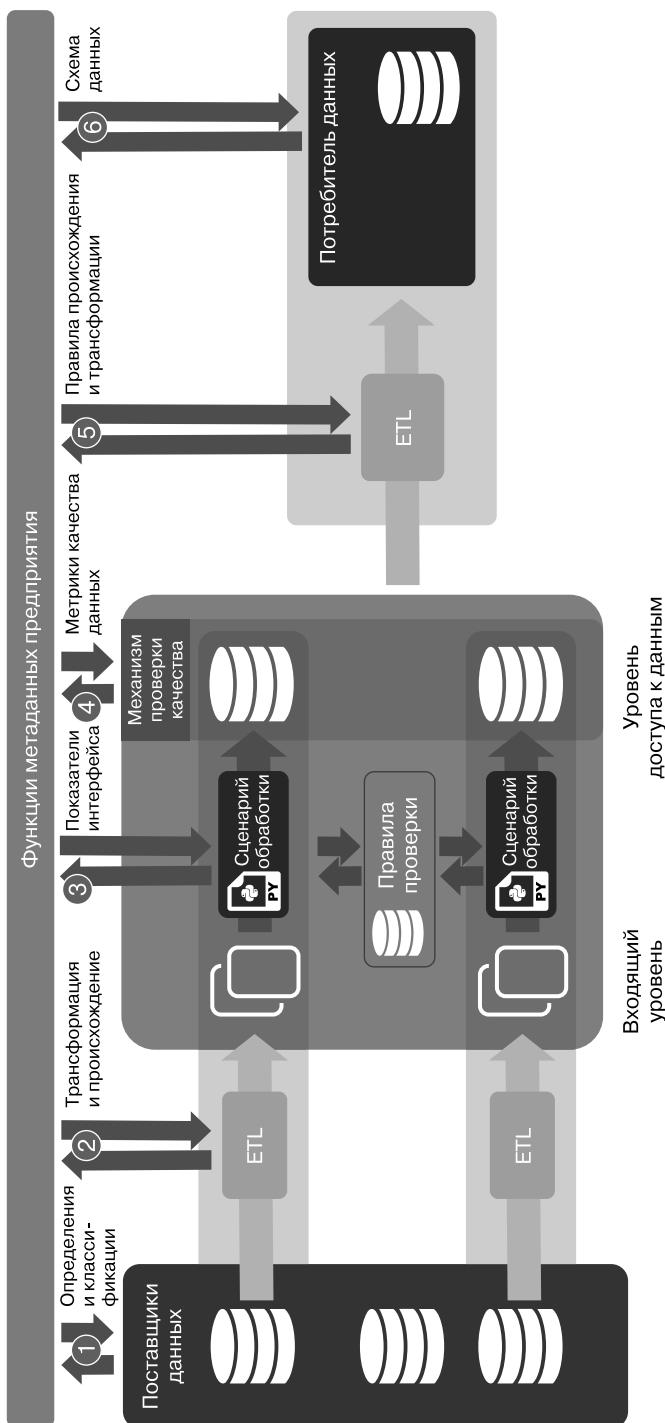


Рис. 3.6. Метаданные генерируются из RDS и множества других функций приложения

Представленное мною решение позволяет извлечь некоторые ключевые детали из платформы RDS.

1. Системы-источники могут применяться для непосредственного получения схемы данных, показателей качества данных (в источнике) и используемых справочных данных. В зависимости от поддерживаемых возможностей можно определить, какие данные находятся в каждой таблице и каждом столбце — например, информация о кредитных картах, именах и адресах клиентов. Чтобы завершить общую картину, вы можете попросить владельцев данных и приложений предоставить дополнительный контекст, например, определения данных и классификации безопасности.
2. Для понимания всех перемещений данных должна быть собрана логика преобразования на этапе интеграции и доставки между источником и RDS. Эти метаданные называются *метаданными происхождения*. Их можно извлечь либо автоматически, используя возможности ETL, либо попросить владельцев системы-источника предоставить их. Логично, что для этого требуются дополнительные компоненты, которые облегчают работу поставщиков данных.
3. На этапе обработки данных вы можете получить информацию о состоянии каждого интерфейса. Если схемы данных отсутствуют или данные технически недействительны, они всегда будут отклоняться. Вы можете проверить эти сведения по контрактам на поставку (см. подраздел «Контракты на поставку данных и соглашения о совместном их использовании» на с. 63), чтобы увидеть, на каких потребителей влияют плохо работающие интерфейсы и каких поставщиков данных необходимо уведомить.
4. Для получения таких характеристик, как точность, полнота и согласованность данных, можно использовать механизмы оценки качества. И снова, если доставляются некачественные данные, файлы, записи или события, их следует отложить в сторону или отклонить и уведомить поставщиков.
5. Логика преобразования, полученная с помощью инструментов ETL и служб интеллектуальной обработки данных, показывает, что произошло с данными и куда они переместились. Эти метаданные могут собираться автоматически или доставляться самими предметными областями. Чтобы облегчить работу с различными областями, необходимо предоставить дополнительные (самообслуживающиеся) компоненты.
6. Системы потребителей данных можно использовать для получения тех же данных, что и на этапе 1.

Сбор метаданных играет важную роль в управлении данными, потому что позволяет контролировать перемещение данных и их использование в дальнейшем. Связав метаданные с качеством данных, также можно определить влияние некачественных данных на предметные области. Вам будет нужно представить всю эту информацию в удобной для пользователя форме.

Метаданные позволяют добавлять другие варианты и реализации той же архитектуры RDS. Пока метаданные собираются последовательно, вы не потеряете понимание общей архитектуры данных. Мы вернемся к этим аспектам в главе 10.

Качество данных

Состояние и качество данных — еще один важный аспект проектирования. Когда наборы данных загружаются в RDS, они проверяются на соответствие определенным метрикам с использованием правил оценки качества данных. В первую очередь должна оцениваться целостность данных и возможность их технической проверки на соответствие опубликованным схемам. Затем выполняется проверка данных на соответствие таким функциональным показателям качества, как полнота, точность, согласованность, достоверность и т. д.

Преимущество использования общей инфраструктуры для RDS заключается в возможности использовать всю мощь больших данных для оценки качества. Например, Apache Spark (<https://spark.apache.org/>) может проверить и обработать сотни миллионов строк данных в течение нескольких минут. Существуют также фреймворки, которые можно использовать для проверки качества данных. В Amazon Web Services (AWS), например, был разработан Deequ¹ (<https://github.com/awslabs/deequ.com>), — инструмент с открытым исходным кодом для расчета показателей качества данных. Другой пример — инструмент Delta Lake² (<https://delta.io>), разработанный в DataBricks (<https://databricks.com>), — его можно использовать как для проверки схемы, так и для проверки данных. Эти инструменты позволяют глубже понять качество данных, что важно для всех сторон (рис. 3.7).

Если мониторинг качества данных реализован как надо, он не только обнаруживает и отслеживает проблемы с качеством данных, но и становится частью общей структуры управления, которая добавляет новые данные к уже существующим. Если по какой-то причине качество упадет ниже указанного порогового значения, структура может зарегистрировать данные и попросить владельцев рассмотреть и принять данные с текущим качеством или отклонить их и выполнить повторную доставку.

Качество данных можно контролировать в двух местах: в источнике или в RDS. Преимущество управления качеством данных в RDS с использованием нескольких источников данных на одной платформе заключается в возможности проверки ссылочной целостности. Системы-источники часто ссылаются на данные из других приложений или систем. Путем перекрестной проверки ссылочной целостности, а также сравнения и сопоставления всех наборов данных из всех RDS можно найти ошибки и корреляции, о существовании которых и не подозревали.

¹ У AWS есть хорошее руководство по масштабируемому тестированию качества данных с помощью Deequ (<https://oreil.ly/e2ayO>).

² Delta Lake имеет несколько функций для проверки схемы, форматирования и настроек по умолчанию (<https://oreil.ly/-tLuO>).

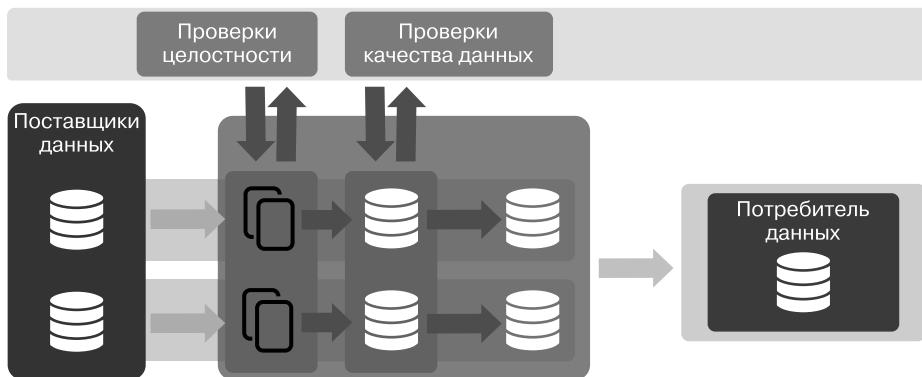


Рис. 3.7. RDS — идеальное место для контроля качества данных. Поскольку здесь собраны разные источники, вы можете проверить ссылочную целостность во многих системах

Оценка качества данных означает наличие в системе управления ими замкнутой цепочки обратной связи, которая постоянно исправляет и предотвращает повторное появление проблем с качеством. Благодаря этому качество данных постоянно контролируется и изменения его уровня должны немедленно устраиваться. Проблемы качества данных должны решаться в системах-источниках, а не в RDS. Если данные исправлены в источнике, то проблемы с качеством больше не будут появляться в других местах.

Мой опыт показывает, что нельзя недооценивать влияние плохого качества данных. Если качество данных не обеспечивается должным образом, все потребители данных будут вынуждены снова и снова проводить работы по их очистке и исправлению.

Уровни RDS

У потребителей могут быть самые разные нужды — от простого исследования данных до принятия решений в режиме реального времени. Чтобы облегчить использование различных схем потребления, я рекомендую разбить RDS на разные уровни: входящий уровень и уровень доступа.

Входящие уровни, как показано слева на рис. 3.8, обычно основаны на недорогом (объектном) хранилище и используются для проверки качества входящих наборов данных, обработки для получения метаданных, а также архивирования и создания наборов исторических данных (вскоре мы обсудим каждую из этих прикладных функций). Естественно, что во входящих уровнях со временем накапляются большие объемы данных; обычно в виде набора файлов CSV или JSON. Наконец, данные могут передаваться на входящий уровень различными способами: пакетной передачей, загрузкой по событиям, загрузкой через API, захватом изменений и т. д.

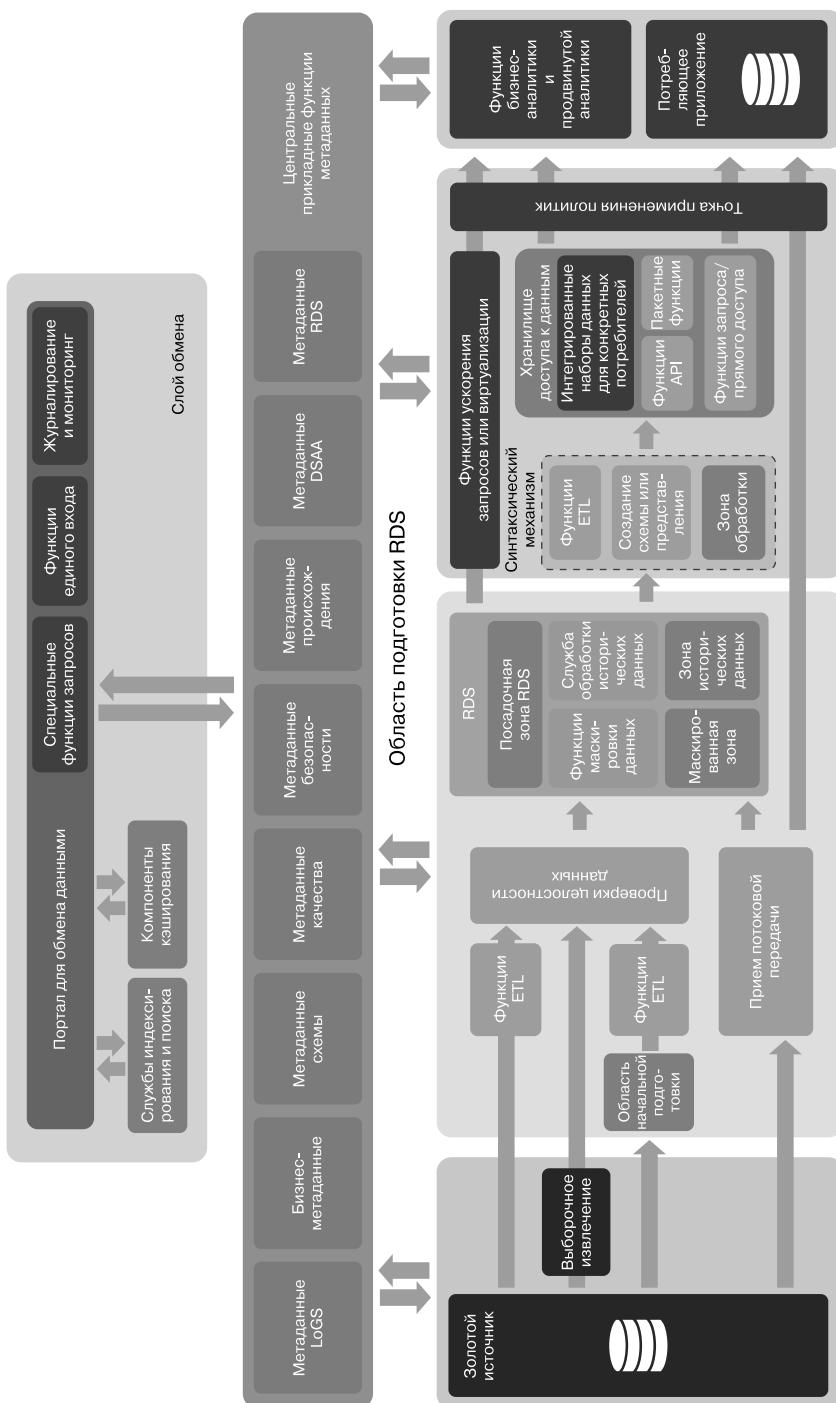


Рис. 3.8. Логическая декомпозиция и детальная разбивка архитектуры RDS. Прием, проверка и управление версиями не связаны с моделями потребления (см. полную версию онлайн: https://oreil.ly/dmas_figs)



RDS используют *разделение*, предполагающее размещение данных от разных поставщиков в разных сегментах, папках или логических экземплярах базы данных. Разделение обеспечивает управляемость и изоляцию.

Уровни доступа к данным, как показано справа на рис. 3.8, оптимизированы для удобства чтения и могут предлагать несколько возможностей обработки запросов для улучшения взаимодействия с пользователем или повышения производительности. Хотя данные все еще находятся в контексте предметной области, они лучше подходят для решения бизнес-вопросов. Очевидно, что этот уровень дороже и часто приходится использовать несколько таких уровней для поддержки различных вариантов использования с разными моделями потребления. Например, для быстрых операций наборы данных могут находиться в хранилищах типа «ключ — значение» или в базах данных в памяти, а текстовые запросы могут располагаться в хранилищах, оптимизированных для поиска. Это также означает, что данные можно преобразовать в форму, которая лучше всего подходит для анализа. Их можно объединять, фильтровать или обрабатывать с использованием исторических данных или только подмножества данных входящего уровня.

Наконец, уровни доступа к данным предоставляют дополнительные возможности. Они могут предлагать самообслуживание и улучшенные средства управления безопасностью, а также инструменты бизнес-аналитики и аналитики для быстрого доступа к данным, и могут использовать метаданные для управления перемещением данных и их синхронизации.

Получение данных

Давайте подробнее рассмотрим входящие уровни и то, как данные принимаются, собираются и накапливаются. Для перемещения данных приложений из золотых источников в RDS используются два основных метода.

Первый — *событийно-ориентированная обработка*, или *загрузка данных по событиям*. Предполагает передачу данных в виде потока небольших событий или наборов данных. Событийно-ориентированная обработка позволяет собирать и обрабатывать данные относительно быстро, поэтому ее часто называют обработкой «в реальном времени». По мере поступления данных могут применяться критерии и преобразования данных для обнаружения изменений в потоке событий (мы коснемся передачи состояния приложения в главе 5 и в частности в разделе «Потоковое взаимодействие» на с. 184). Прием потоковой передачи хорошо подходит для случаев, когда объемы данных относительно малы, вычисления, выполняемые с данными, относительно просты

и задержки должны быть относительно короткими, близкими к реальному времени. Многие сценарии использования допускают применение событийно-ориентированной обработки, но если полнота и объем данных являются серьезной проблемой, то предпочтительнее использовать традиционную пакетную обработку.

Второй метод — *пакетная обработка* — подразумевает процесс одновременной передачи большого объема данных, например всех транзакций из крупной платежной системы в конце дня. Пакет обычно выглядит как набор данных с миллионами записей, хранящихся в виде файла. Почему может понадобиться пакетная обработка? Потому что во многих случаях это единственный способ собрать данные. Наиболее важные RDBM имеют пакетные компоненты для поддержки пакетного перемещения данных. *Согласование данных*, процесс проверки данных во время перемещения, часто вызывает беспокойство¹. Проверка наличия всех данных важна, особенно в финансовых процессах. Кроме того, многие системы сложны и содержат огромное количество таблиц, и применение языка структурированных запросов (Structured Query Language, SQL) — единственный надежный способ выбрать и извлечь все необходимые данные. Вот почему такие утверждения, как «пакеты не нужны» или «событийно-ориентированные данные — единственный вариант», слишком обобщены.

Хотя загрузка данных по событиям продолжает набирать популярность, этот метод никогда не сможет полностью заменить пакетную обработку во всех случаях использования. Я предполагаю, что на современном предприятии одновременно будут использоваться оба метода. Еще ожидается, что для предметных областей будет реализовано много дополнительных компонентов предоставления данных, учитывая большое разнообразие приложений и БД. Они будут различаться в зависимости от особенностей процессов ETL, захвата изменений в данных, средств планирования и т. д.

Метаданные схемы

При пакетной обработке старых систем одним из слабых мест обычно является управление метаданными схемы. Для обеспечения целостности и полноты данных в файлах TXT и CSV (значения, разделенные запятыми) также можно добавить вверху заголовок, описывающий схему и количество строк внизу в файле CSV. Другой метод описания файлов — разработать определение интерфейса

¹ Согласование (проверка данных) может быть реализовано разными способами. Можно сравнивать количество строк, использовать контрольные суммы столбцов (<https://oreil.ly/NqCeO>), разделять большие таблицы и повторно обрабатывать их, а также использовать инструменты сравнения данных, такие как Redgate (<https://www.redgate.com/>).

с использованием метаданных схемы¹. Файл метаданных, который показан в примере 3.1, описывает не только схему данных, но и владение, контрольные суммы и версию. Контрольные суммы могут использоваться для проверки полноты после преобразования данных, а версия — для проверки эволюции и обратной совместимости.

Пример 3.1. Пример файла схемы или дескриптора, который можно использовать для описания доставки данных

```
<?xml version="1.0" encoding="UTF-8"?>

<meta xmlns="http://www.w3.org/ns/csvw">
  <owner id="63845">
    <contact>John Snow - johnsnow@example.com</contact>
    <department>HR</department>
  </owner>
  <file-location format="csv" compression="gzip" separator="\t">
    <chunk location="data_1.csv.gz" size="123456" checksum="8014462f532"/>
    <chunk location="data_2.csv.gz" size="34354" checksum="6f5f0793aab"/>
    <!-- объявления других фрагментов ... -->
    <chunk location="data_N.csv.gz" size="7890123" checksum="5beede7a808"/>
  </file-location>
  <classifications>
    <label>PERSONAL_DATA</label>
    <label>HR</label>
  </classifications>
  <data-fields>
    <string-field name="ID" description="Unique identification number"/>
    <string-field name="USERNAME" description="Company username"/>
    <string-field name="FIRSTNAME" sensitive="YES" description="Firstname"/>
    <string-field name="LASTNAME" sensitive="YES" description="Lastname"/>
    <categorical-field name="GENDER" sensitive="YES" description="Gender"/>
      <category value="F"/>
      <category value="M"/>
    </categorical-field>
    <continuous-field name="AGE" missing="0" description="Age in years"/>
    <categorical-field name="COUNTRY" missing="?" description="Country of residence">
      <category value="USA"/>
      <category value="UK"/>
      <category value="OTHERS"/>
    </categorical-field>
  </data-fields>
</meta>
```

¹ У Маттиаса Фридриха (Matthias Friedrich) есть подобный пример (<https://oreil.ly/XQrJV>).

Еще один подход к предоставлению метаданных с данными — разрешение областям регистрировать свои интерфейсы и выгружать свои метаданные на центральный портал регистрации. Этот реестр метаданных можно использовать и для хранения информации об интерфейсах, данных и связанных концепциях в одном месте. Мы обсудим это в главе 6.

Я уделяю так много внимания метаданным, потому что они важны как для совместимости, так и для поддержки передачи данных в хранилища RDS. В отсутствие метаданных вам придется вручную разрабатывать множество конвейеров¹. Метаданные помогут автоматизировать обработку и организовать создание дополнительных контрольных точек проверки, работающих согласованно, независимо от используемой базовой технологии.

В следующих двух разделах мы рассмотрим некоторые продукты и службы, требующие дополнительного внимания для правильного приема данных. Сюда входит сбор данных из готовых коммерческих решений и внешних сервисов.

Интеграция готовых коммерческих решений

Дополнительного внимания требуют интеграция и сбор данных из готовых коммерческих (commercial off-the-shelf, COTS) продуктов. Многие из них чрезвычайно трудно интерпретировать или использовать. Схемы БД часто бывают очень сложными, а ссылочная целостность обычно поддерживается программно, через приложение, а не базу данных. Часто данные защищены и могут извлекаться только с помощью стороннего решения.

Во всех ситуациях я рекомендую реализовать дополнительные службы, которые позволят сначала выгрузить данные во вторичное хранилище (рис. 3.9), а затем настроить конвейер для переноса данных из этого хранилища в RDS. Преимущество такого подхода в том, что решение поставщика отделено от конвейера данных. Обычно схема данных продукта COTS напрямую не контролируется. Если поставщик выпустит обновление продукта и изменит структуры данных, то конвейер данных перестанет работать. Подход с промежуточным хранилищем обеспечивает гибкость, позволяющую поддерживать совместимость интерфейса.

¹ Конвейер данных — это обобщенный термин, обозначающий обработку и перемещение данных от источника к месту назначения. Конвейер может обрабатывать данные как в реальном времени, так и в пакетном режиме и при необходимости включать логику преобразования.



Рис. 3.9. При использовании комплексных решений COTS лучше выгрузить данные во вторичное хранилище и уже оттуда переносить их в RDS

Извлечение данных из внешних API и SaaS

Внешние API или SaaS, играющие роль поставщиков, тоже обычно требуют особого внимания. Бывают случаи, когда нужно получить полный набор данных, но API позволяет извлечь только относительно небольшую часть. Другие API могут применять регулирование, используя квоты или ограничения на количество запросов. Есть также API с дорогими тарифными планами для каждого вашего вызова.

Во всех этих ситуациях я рекомендую создавать небольшие службы или приложения, которые обращаются к API и хранят данные во вторичном хранилище, как показано на рис. 3.10.



Рис. 3.10. Для извлечения данных из решения SaaS лучше использовать обертки вокруг API

Создавая обертки, инкапсулирующие API поставщика SaaS, можно спроектировать интересный шаблон. Все вызовы сначала будут передаваться обертке, инкапсулирующей API. Если запрос был выполнен недавно или только что, то обертка немедленно вернет результаты из вторичной БД. В противном случае будет вызван API поставщика SaaS, а результаты переданы потребителю и одновременно сохранены во вторичном хранилище для любого последующего использования. С помощью этого шаблона можно в конечном итоге получить полный набор данных и заполнить хранилище данных только для чтения (RDS).

Служба исторических данных

Аспект, который я хочу обсудить более подробно, — это управление жизненным циклом данных путем сбора и хранения исторических данных. Удаление нерелевантных данных делает системы более быстрыми и экономичными.

Архитектура RDS берет на себя роль хранения и управления большими объемами исторических данных. Основное отличие от архитектуры корпоративного хранилища состоит в том, что RDS хранят данные в исходном контексте. Никакого преобразования в модель данных предприятия не ожидается, поэтому ценность не будет потеряна. Это серьезное преимущество: в оперативных сценариях использования, требующих сохранения большого количества исторических данных, не нужно преобразовывать данные обратно в исходный контекст.

Хотя RDS не зависит от технологии, вероятность того, что все входящие уровни RDS будут спроектированы с использованием дорогостоящих систем управления AWS, очень мала. Когда хранилище и вычислительные ресурсы разделены, ваши RDS, скорее всего, будут размещены в недорогих распределенных файловых системах, доступных только для добавления. Это означает, что любое изменение или обновление существующих таблиц повлечет за собой полное переписывание файлов или уровней доступа к данным. Поэтому для распределенных файловых систем, доступных только для добавления, я советую использовать один из описанных ниже подходов, так как существует компромисс между затратами в управление входящими данными и простотой их потребления. У каждого подхода есть свои плюсы и минусы.

Разбиение на разделы полноразмерных моментальных снимков файловых систем

Первый подход — сохранять все доставляемые данные путем логического разбиения на разделы и группировки. Разбиение на разделы — распространенный метод организации файлов или таблиц в отдельные группы (разделы) для повышения управляемости, производительности или доступности¹. Разбиение обычно выполняется по некоторым атрибутам данных, таким как географическое местоположение (город, страна), значения (уникальные идентификаторы, коды сегментов) или время (дата и время доставки). Пример разбиения на разделы показан на рис. 3.11, слева.

¹ Разбиение на разделы полных моментальных снимков данных можно идеально сочетать с недорогими распределенными файловыми системами, такими как HDFS (<https://oreil.ly/PfsZi>), S3 (<https://aws.amazon.com/s3>), ADLS (<https://oreil.ly/lmnyJ>) и т. д. В Datio есть пример использования HDFS (<https://oreil.ly/YZpZN>).

Данные разделены по дате доставки

id	name	e-mail	delivery_date
1	John Snow	john.snow@example.com	1-1-2019
2	Brian Stark	brian.stark@example.com	1-1-2019
1	John Snow	john.snow@example.com	2-1-2019
2	Brian Stark	brian.stark@example.com	2-1-2019
3	Elis Smith	elis.smith@example.com	2-1-2019
1	John Snow	john.snow@example.com	3-1-2019
2	Brian Stark	brian.stark@example.com	3-1-2019
3	Elis Smith	elis.smith@example.com	3-1-2019

Данные обрабатываются и сохраняются в медленно меняющихся измерениях (тип 2), что позволяет пользователям выбирать конкретные значения

id	name	e-mail	start_date	end_date
1	John Snow	john.snow@example.com	1-1-2019	
2	Brian Stark	brian.stark@example.com	1-1-2019	
3	Brian Stark	brian.stark@example.com	3-1-2019	
3	Elis Smith	elis.smith@example.com	3-1-2019	

Рис. 3.11. Наглядные примеры, как будут выглядеть данные при разделении на раздели на использование полноразмерных моментальных снимков или медленно меняющихся измерений (тип 2)

Разбиение полных моментальных снимков на разделы выполнить проще. По мере поступления данных каждый снимок добавляется как новый неизменяемый раздел. Таблица — это набор всех снимков, в которых каждый раздел сохраняет полный размер на определенный момент времени. Недостаток такого подхода — дублирование данных. Я не считаю это проблемой в наши дни, когда облачные хранилища стоят относительно дешево. Полные моментальные снимки также упрощают повторную доставку. Если исходная система обнаруживает, что были доставлены неверные данные, их можно отправить снова и раздел будет перезаписан. Основной недостаток этого подхода — усложнение анализа данных. Сравнение между конкретными периодами времени может быть затруднено из-за необходимости обрабатывать все данные при чтении. Это может стать проблемой, если потребители требуют, чтобы все исторические данные были обработаны и сохранены. Обработка исторических данных за три года может привести к последовательной обработке как минимум тысячи файлов и занять много времени, в зависимости от размера данных.

Обслуживание исторических данных

Второй подход — оптимизация всех наборов данных для использования исторических данных. Например, обработка всех наборов данных в SCD¹, которые показывают все изменения, имевшие место с течением времени. Обработка и создание исторических данных требуют дополнительного процесса ETL для обработки и объединения различных доставок данных (см. правую часть рис. 3.11).

Подход к созданию исторических данных позволяет организовать их хранение более эффективно, обрабатывая, удаляя дубликаты и объединяя данные. Как можно видеть на рис. 3.11, медленно меняющееся измерение занимает половину количества строк. Поэтому запросы, например, при использовании реляционной БД выглядят проще и выполняются быстрее. Очистка данных или удаление отдельных записей, что может потребоваться для соблюдения требований GDPR, также упростится, так как вам не придется обрабатывать все наборы данных. Еще одно преимущество — возможность выбора более простых и быстрых реляционных БД благодаря более эффективному хранению данных.

Однако есть и недостаток: создание SCD требует большего управления. Все данные нужно обработать. Изменения в исходных данных необходимо обнаруживать сразу после их появления и затем обрабатывать. Это требует дополнитель-

¹ Медленно меняющееся измерение (slowly changing dimension, SCD) — это измерение, которое хранит как текущие, так и исторические данные в хранилище и управляет ими. Таблицы измерений в управлении данными и хранилищах данных могут поддерживаться с помощью нескольких методологий, называемых типами от 0 до 6. Различные методологии описаны Kimball Group (<https://oreil.ly/NUUW1>).

тельного кода и вычислительной мощности. Еще хочу заметить, что потребители данных имеют разные требования. Поэтому, несмотря на наличие медленно меняющихся измерений, потребителям все равно необходимо обрабатывать данные. Например, данные могут поступать и обрабатываться каждый час, но, если потребитель ожидает, что данные будут сравниваться по дням, все равно потребуется дополнительная работа по преобразованию.



Один из недостатков создания исторических данных для общего потребления состоит в том, что потребителям все равно может понадобиться обрабатывать данные всякий раз, когда они пропускают столбцы в выборках. Например, если потребитель запрашивает более узкий набор данных, могут появиться повторяющиеся записи и придется снова выполнить обработку для удаления дубликатов.

Наконец, повторная доставка может быть трудновыполнимой, так как при этом могут быть обработаны и добавлены в измерения некорректные данные. Это можно исправить с помощью логики повторной обработки, дополнительных версий или сроков действия, но в любом случае потребуется дополнительное управление. Проблема здесь в том, что правильное управление данными может стать обязанностью центральной группы, поэтому такая масштабируемость нуждается в обширном интеллектуальном самообслуживающемся функционале.

Для масштабируемости и помощи потребителям вы также можете рассмотреть возможность смешивания двух подходов — сохранение всех полных моментальных снимков и создание «исторических данных как услуги». В этом сценарии всем предметным областям потребителей предлагается небольшая вычислительная инфраструктура, с помощью которой они смогут планировать создание исторических данных в зависимости от объема (ежедневно, еженедельно или ежемесячно), временного интервала, атрибутов и необходимых им наборов данных. Используя краткосрочные экземпляры обработки в общедоступном облаке, вы даже можете сделать такой подход рентабельным. Большим преимуществом этого решения является сохранение гибкости при повторной доставке при отсутствии необходимости привлекать команду инженеров для работы с данными. Еще одно преимущество — возможность адаптировать временные рамки, объемы и атрибуты к потребностям каждого клиента.

Измерения только для добавления

Другой подход к сбору и хранению исторических данных — использование способа доставки только с добавлением. При этом из базы данных приложения загружаются только новые или измененные записи и добавляются в существующий набор записей. Доставка только с добавлением хорошо подходит

для транзакционных данных, а также для данных на основе событий, которые мы обсудим в главе 5.

Измерения только для добавления можно сочетать с разбиением данных на разделы и медленно меняющимися измерениями. Основные данные, например, могут доставляться из приложения и собираться с использованием способа медленно меняющихся измерений, а транзакционные данные для той же системы — обрабатываться методом доставки только с добавлением.

Повторные доставки и поздно прибывающие данные

Для повторных доставок и поздно прибывающих данных измерения должны разбиваться на разделы на основе времени появления поставщика данных, а не времени обработки RDS. Для повторных доставок можно рассмотреть возможность включения времени обработки в качестве дополнительного столбца¹. В этом случае потребители смогут выбрать любую из доставок.

Варианты проектирования

Давайте переключимся на сторону потребителя в архитектуре RDS и посмотрим, какие компоненты и сервисы нужно предоставить. Потребление данных из уровней доступа к данным имеет свои требования к обработке, влияющие на организацию RDS. Организация RDS в значительной степени зависит от вариантов использования и требований областей-потребителей. Во многих случаях к обработке данных не предъявляется жестких требований по времени. Иногда ответ на вопрос или удовлетворение бизнес-потребности может подождать несколько часов или даже несколько дней. Но иногда данные должны доставляться в течение нескольких секунд.

Мы стали свидетелями увеличения разнообразия систем БД. Обычно транзакционные и операционные системы рассчитаны на обеспечение высокой целостности данных и поэтому используют ACID-совместимые и реляционные БД. Но базы данных, не соответствующие принципам ACID и не имеющие схемы, тоже актуальны, потому что они ослабляют строгость контроля целостности в пользу более высокой производительности. А еще благодаря им можно использовать более гибкие типы данных: данные, поступающие с мобильных телефонов, игр, каналов социальных сетей, датчиков и т. д. Популярные БД для таких типов взаимодействий — это обычно хранилища документов или пар «ключ — значение». Наконец, размер данных тоже может иметь значение.

¹ Ральф Кимбалл (Ralph Kimball) в своей книге *The Data Warehouse Toolkit* назвал этот метод «непредсказуемыми изменениями с наложением одной версии». Эта методология также известна как «SCD 6-го типа».

Одни приложения хранят относительно небольшой объем данных, а другим могут понадобиться терабайты информации.

ПОЧЕМУ СУЩЕСТВУЕТ ТАК МНОГО РАЗНЫХ БАЗ ДАННЫХ

При разработке БД важно учитывать большое количество факторов и компромиссов. Есть структуры данных. Есть компромиссы между согласованностью, доступностью и возможностью разбивания. Есть кэширование и индексирование для повышения эффективности поиска. Есть разные способы хранения и извлечения данных: небольшими фрагментами, большими фрагментами, отсортированными фрагментами и т. д. Есть компромиссы, связанные с распределением и согласованностью, которые могут повлиять на производительность. Есть такие функции, как постоянный мониторинг и аналитика, которые также влияют на производительность. Наконец, есть и нефункциональные требования: привязка к поставщику, поддержка, совместимость и языки запросов. Суть в том, что ни одна БД не может превзойти другие по всем параметрам одновременно. Выбирайте базу данных, которая лучше соответствует вашим требованиям.

Сложно обрабатывать разные формы данных при использовании корпоративного хранилища для их распространения. Корпоративные хранилища обычно проектируются с использованием единой технологии (движка базы данных RDBM) и не оставляют много места для вариаций.

Преимущество архитектуры RDS в том, что мы можем добавить несколько разновидностей RDS для определенных шаблонов чтения в разных вариантах использования. Из-за того, что данные разрешено дублировать, можно одновременно поддерживать разные структуры данных, скорости и объемы. Это означает, что данные поставщика можно скопировать в несколько RDS, размещенных на разных платформах, чтобы предоставить потребителям различные измерения и удовлетворить требования разных вариантов использования. Во всех случаях изменения контекста не учитываются, поэтому используются одни и те же данные, представленные в разных формах. На рис. 3.12 показан гипотетический пример, иллюстрирующий поддержку различных вариантов использования.

Поставщик на рис. 3.12 использует три различных RDS, которые хранят данные и передают их потребителям. Одни и те же данные дублируются с использованием разных представлений. Конечная точка, основанная на реляционных технологиях, должна облегчить выполнение более сложных и непредсказуемых запросов. Конечная точка на основе столбцов упрощает работу потребителям, широко использующим агрегирование данных, а конечная точка на основе документов предназначена для потребителей, которым требуется более высокая скорость и полуструктуренный формат данных (JSON). Все три конечные точки управляются данными и должны быть построены с использованием одних и тех же метаданных. Ответственность за данные лежит на одной и той же области-

поставщике. Принципы многократно используемых и оптимизированных для чтения данных также применимы ко всем вариациям RDS. Кроме того, ожидается, что контекст и семантика будут оставаться неизменными. Например, понятие «клиент» (или «имя клиента») должно быть одинаковым во всех трех вариациях. Один и тот же единый язык используется для всех RDS, принадлежащих одному поставщику данных. Обеспечение смысловой согласованности всех конечных точек RDS и других шаблонов интеграции мы обсудим в главе 6.

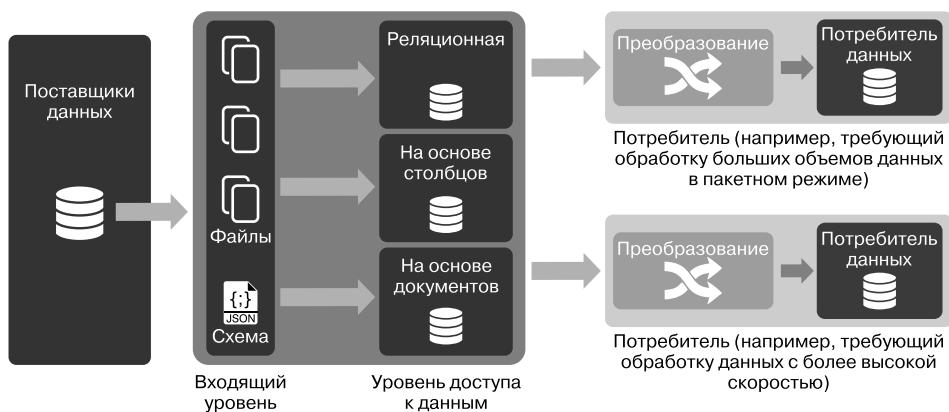


Рис. 3.12. В рамках архитектуры RDS можно разрешить областям предоставлять одни и те же данные через несколько конечных точек RDS, используя разные представления. При таком подходе области могут поддерживать и оптимизировать разные измерения и варианты использования

Репликация данных

Другой аспект хранилищ данных только для чтения — репликация в распределенной среде. Она необходима для синхронизации и актуализации различных экземпляров и уровней RDS. Реальность такова, что области распределены, разбросаны и размещены на разных узлах. Поскольку во многих ситуациях сеть — ограничивающий фактор, нужно избегать многоократного извлечения одинаковых данных и создания слишком большого сетевого трафика. В идеале данные должны передаваться по сети только один раз, а потом распределяться внутри по разным областям. Таким образом, RDS необходимы для передачи данных другим RDS. На рис. 3.13 показано, как можно организовать распределение данных между различными RDS.

Не все данные нужно копировать. Воспроизводите их только тогда, когда это требуется для областей. Соглашения о совместном использовании данных

(рис. 3.13), которые хранятся в хранилище метаданных, содержат информацию о том, какие данные в каком месте используются. Они помогают определить, какие данные подлежат синхронизации или копированию.

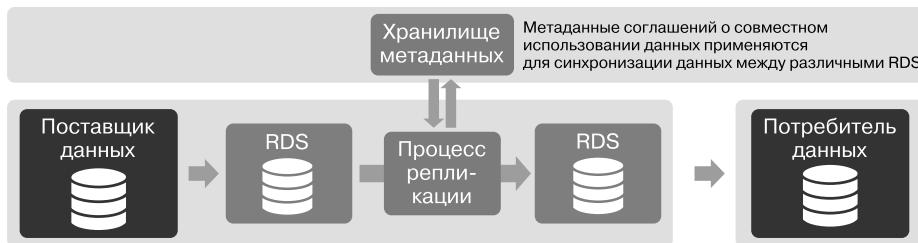


Рис. 3.13. Синхронизация RDS позволяет решить проблемы с задержкой в сети. Метаданные и соответствующие соглашения о совместном использовании определяют, какие данные необходимо скопировать

Синхронизация данных зависит от выбранной технологии. Для файлов и распределенных файловых систем чаще выбирают Apache Sqoop (<https://sqoop.apache.org/>). Другие популярные варианты с открытым исходным кодом — rsync (<https://rsync.samba.org/>) и rclone (<https://rclone.org/>). В числе коммерческих предложений можно упомянуть продукты от основных поставщиков облачных услуг, таких как Azure StorSimple (https://oreil.ly/w_BRa) и AWS Storage Gateway (<https://oreil.ly/IWeby>). WANdisco (<https://www.wandisco.com/>) — еще одна компания в этой сфере, предлагающая универсальное облачное решение для репликации. Коммерческие решения копируют данные на более детальном уровне, не заставляя писать много кода.

Для синхронизации в реальном времени можно использовать событийно-ориентированную архитектуру, которая копирует данные при их изменении. Этот шаблон мы рассмотрим в главе 5.

Уровень доступа

Для организации взаимодействий пользователей и приложением с RDS следует предусмотреть большое разнообразие шаблонов доступа к данным. Сюда могут входить специальные запросы, прямое представление отчетов, создание семантических слоев или кубов, поддержка открытого доступа к БД (Open Database Connectivity, ODBC) для других приложений, обработка с помощью инструментов ETL, преобразование или обработка данных.

Для поддержки всего этого важно обеспечить достаточную производительность RDS и реализовать функции безопасности. В зависимости от выбранной

технологии хранения и предоставления данных может понадобиться добавить в архитектуру дополнительный уровень доступа (рис. 3.14).



Рис. 3.14. Дополнительный уровень доступа или механизм запросов, например Presto или Azure Synapse, помогает предотвратить создание ненужных копий данных



Уровень доступа к данным можно реализовать как слой доступа. В этом случае уровень доступа к данным виртуализирован (<https://oreil.ly/XdKhe>).

Применение распределенных файловых систем, таких как HDFS с Apache Hive (<https://hive.apache.org/>), экономически оправданно для хранения больших объемов данных, но обычно они недостаточно быстры для специальных и интерактивных запросов. В Facebook, например, осознали это и разработали Presto (<https://prestodb.github.io>) для решения проблемы. Apache Drill (<https://drill.apache.org/>), Apache Kylin (<http://kylin.apache.org/>) и Apache Arrow (<https://arrow.apache.org/>) — похожие проекты с открытым исходным кодом, преследующие одну и ту же цель: поддержка интерактивных и аналитических запросов с помощью высокопроизводительного механизма запросов. Подобные варианты доступны в облаке: AWS Athena (<https://oreil.ly/t758>), Azure Synapse (<https://oreil.ly/neh96>) и Google Dataproc (<https://oreil.ly/HSRMW>).

Преимущество выполнения запросов напрямую к RDS заключается в том, что данные не нужно дублировать. Это делает архитектуру дешевле. Кроме того, эти инструменты предлагают детальный доступ к данным, и, наконец, они способны заместить операционное хранилище данных. RDS становится идеальным местом для оперативного получения отчетов и специального анализа.

Служба обработки файлов

Продолжая обсуждение потребления данных, отметим, что в некоторых областях поддерживаются варианты использования или приложения, принимающие только плоские файлы (например, CSV). Для таких случаев можно создать службы обработки файлов (рис. 3.15), автоматически удаляющие конфиденциальные данные, которые потребители не должны видеть. Типичный пример — удаление столбцов, фильтрация строк на основе значений полей или изменение значений конфиденциальных полей.

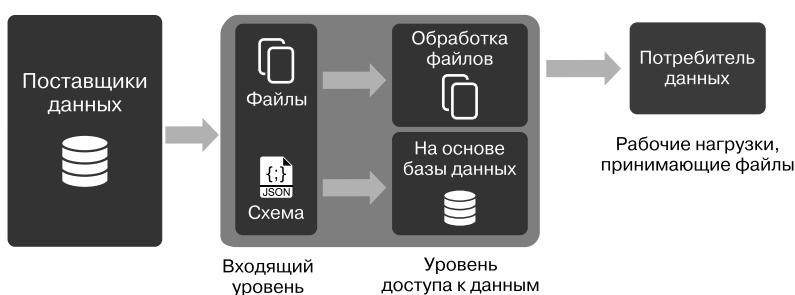


Рис. 3.15. Служба обработки файлов поможет реализовать ограничения безопасности для файлов CSV

Служба, как и все другие конечные точки потребления, должна быть подключена к центральной модели безопасности, обеспечивающей соответствие потребляемых данных соглашениям о совместном использовании (см. подраздел «Контракты на поставку данных и соглашения о совместном их использовании» на с. 63). Она также обеспечивает автоматическое применение фильтров, основанных на классификации и атрибутах метаданных. Мы обсудим это подробно в главе 7.

Служба уведомлений о доставке

Для всех моделей приема и потребления служба уведомлений, позволяющая другим областям автоматизировать и запускать свои рабочие процессы, является одним из основных компонентов, которые должна иметь общая платформа RDS. Многие поставщики облачных услуг предлагают эту функцию, отправляя события всякий раз, когда файлы создаются, открываются, изменяются или удаляются. Кроме того, для отправки таких сообщений и событий можно использовать событийно-ориентированную архитектуру, о которой мы поговорим в главе 5.

Служба удаления персональной информации

При использовании RDS для исследования или анализа данных, машинного обучения и обмена данными с третьими сторонами важно защитить конфиденциальные данные от потребителей. Благодаря таким методам, как токенизация, хеширование, скремблирование и шифрование, можно использовать правила и классификации для защиты данных. В зависимости от организации RDS можно также применить следующие методы.

- Защиту данных во время их приема на собственной платформе. AWS рекомендует использовать обезличенное озеро данных (<https://oreil.ly/BZBpi>), которое заменяет важные и конфиденциальные данные токенами. Это позволяет обезопасить исходные данные.
- Дублирование и обработку данных отдельно для каждого проекта. Вы можете настроить защиту, указав используемые классификации и правила.
- Защиту данных во время потребления. Этот метод защищает конфиденциальные данные, не затрагивая хранимых данных и изменяя только результаты запросов. Microsoft, например, использует функцию, называемую динамической маскировкой данных (<https://oreil.ly/sZ371>), для сокрытия данных при их извлечении из базы.

Безопасность и классификации данных зависят от управления ими. Эти две дисциплины будут более подробно раскрыты в главе 7.

Распределенная оркестрация

Последний аспект, требующий рассмотрения и стандартизации, — это поддержка реализации, тестирования и управления конвейеров данных к RDS и от них. Построение и автоматизация этих конвейеров выполняется в несколько итераций, включающих такие шаги, как извлечение, подготовка и преобразование данных. Вы должны позволить командам тестировать и контролировать качество всех конвейеров данных и артефактов, а также поддерживать их изменениями кода в процессе развертывания. Этот подход к непрерывной доставке очень похож на DataOps, поскольку включает в себя множество средств автоматизации, технических приемов, рабочих процессов, архитектурных шаблонов и инструментов¹.

Apache Airflow (<https://oreil.ly/9xhKB>) — один из лучших инструментов управления рабочими нагрузками и конвейерами данных; он имеет обширную систему управления рабочим процессом и поддерживает популярный язык Python.

¹ Длинный список инструментов и фреймворков поддерживается в блоге DataOps (https://oreil.ly/8_RyF).

Поскольку он в основном предназначен для управления перемещением данных, его часто комбинируют с Apache Spark для выполнения сложных проверок и преобразований. Data Build Tool (DBT) (<https://www.getdbt.com/>) — еще одно популярное решение, оно полностью основано на командной строке и использует операторы SQL для поддержки процессов ETL. BMC Control-M (<https://oreil.ly/Us473>) — это коммерческий и вариант корпоративного уровня с большим количеством адаптеров и возможностями регистрации. Типичные облачные варианты — AWS Batch (<https://oreil.ly/8D8b2>), Dataflow (<https://oreil.ly/Dk68r>) и Azure Data Factory (<https://oreil.ly/S83TT>).

Для тестирования качества и развертывания имеется еще более широкий выбор: DataKitchen (<https://www.datakitchen.io/>) — средство для мониторинга и профилирования качества данных, iCEDQ (<https://icedq.com/>) можно использовать для тестирования; Jenkins (<https://jenkins-ci.org/>) — для развертывания на нескольких платформах; Redgate (<https://www.red-gate.com/>) — это SQL-инструмент, который помогает создавать версии схем данных и формировать новые БД; а Unravel (<https://oreil.ly/I4z19>) можно использовать для управления производительностью.

При стандартизации инструментов и передовых практик я рекомендую организациям создать централизованную или платформенную группу, поддерживающую другие команды с помощью задач, расписания, метаданных, показателей и т. д. Эта группа должна нести ответственность за непрерывный мониторинг и вмешиваться, если конвейеры остаются неработоспособными слишком долго. Чтобы абстрагировать сложности зависимостей и различия во временных интервалах от разных поставщиков данных, эта группа может настроить дополнительную инфраструктуру обработки, которая будет, к примеру, информировать потребителей, когда можно объединить несколько источников с отношениями зависимости.

Интеллектуальные службы потребления

Используя все это, вы можете расширить RDS с помощью интеллектуальных служб потребления, чтобы упростить автоматизацию простых синтаксических преобразований и обмена данными. Эти службы полагаются на уровень метаданных, включая множество прикладных функций, созданных для управления данными. Здесь все становится сложнее, потому что все, что мы обсуждали до сих пор, сходится воедино. Давайте начнем с иллюстрации и будем продвигаться дальше.

После доставки данных потребителям необходим удобный способ выбора и использования данных с помощью интеллектуальных служб интеграции. Представляемые вами шаги преобразования должны быть простыми и не создавать новых данных: подумайте об изменении формата, простом поиске, фильтрах, сопоставлении, простом соединении или объединении, переименовании полей

и обработке исторических данных. Такие манипуляции с данными считаются дополнительными и не дублируют их без необходимости. Это важно, потому что в архитектуре не должно быть слишком много неконтролируемых копий данных.

Если вы посмотрите на схему архитектуры на рис. 3.16, то заметите несколько особенностей. Во-первых, приложения золотых источников передают наборы данных с помощью нескольких шаблонов доставки и приема, таких как пакетная обработка, сбор измененных данных, прием по событиям и получение данных из API. Предоставленные метаданные должны описывать интерфейсы и данные. Во время этого процесса интегрированные предметные области могут поддерживаться дополнительными возможностями. Инструменты автоматического профилирования могут предсказать, какие классификации и определения нужно применить к данным. Портал регистрации метаданных, включая несколько API, может позволить областям самостоятельно поддерживать информацию. Некоторые из этих аспектов будут рассмотрены в главе 6.

После успешной доставки данных качество будет подтверждено и данные будут сохранены в RDS. После этого данные станут широкодоступными, в том числе и через операционную архитектуру (обрабатывающую запросы) в той же предметной области. Это означает, что CQRS применяется как в операционной, так и в аналитической сфере использования данных. Если CQRS сочетается с аспектами транзакционного чтения, приложение должно проектироваться с учетом модели конечной согласованности.

Данные, доступные в RDS, становятся доступными и для потребителей: конечных пользователей и приложений. Как ожидается, конечные пользователи и приложения будут получать их не напрямую, а использовать для этого интуитивно понятный портал обмена данными, который поможет им сориентироваться, а затем порекомендует и автоматически найдет данные. В этом случае потребители смогут выбрать, какие именно наборы данных и элементы они хотели бы использовать и какие простые шаги преобразования применить к данным. Такой подход позволяет потребителям разрабатывать и потреблять данные без необходимости быть экспертами по проектированию этих данных.

Когда потребители сделают свой окончательный выбор, в игру вступят дополнительные компоненты безопасности, после чего поставщикам данных будет предложено изучить и подтвердить запрос потребителя. После получения всех одобрений и заключения соглашения данные станут доступны с помощью одного из шаблонов в правой части рис. 3.16. Как видите, здесь есть хранилище доступа к данным (уровень доступа к данным RDS), которое помогает обслуживать выбор данных RDS для конкретного потребителя. Это хранилище доступа к данным, как мы уже знаем, может быть виртуальным или невиртуальным. Данные также могут быть замаскированы, скремблированы или скрыты для определенных потребителей.

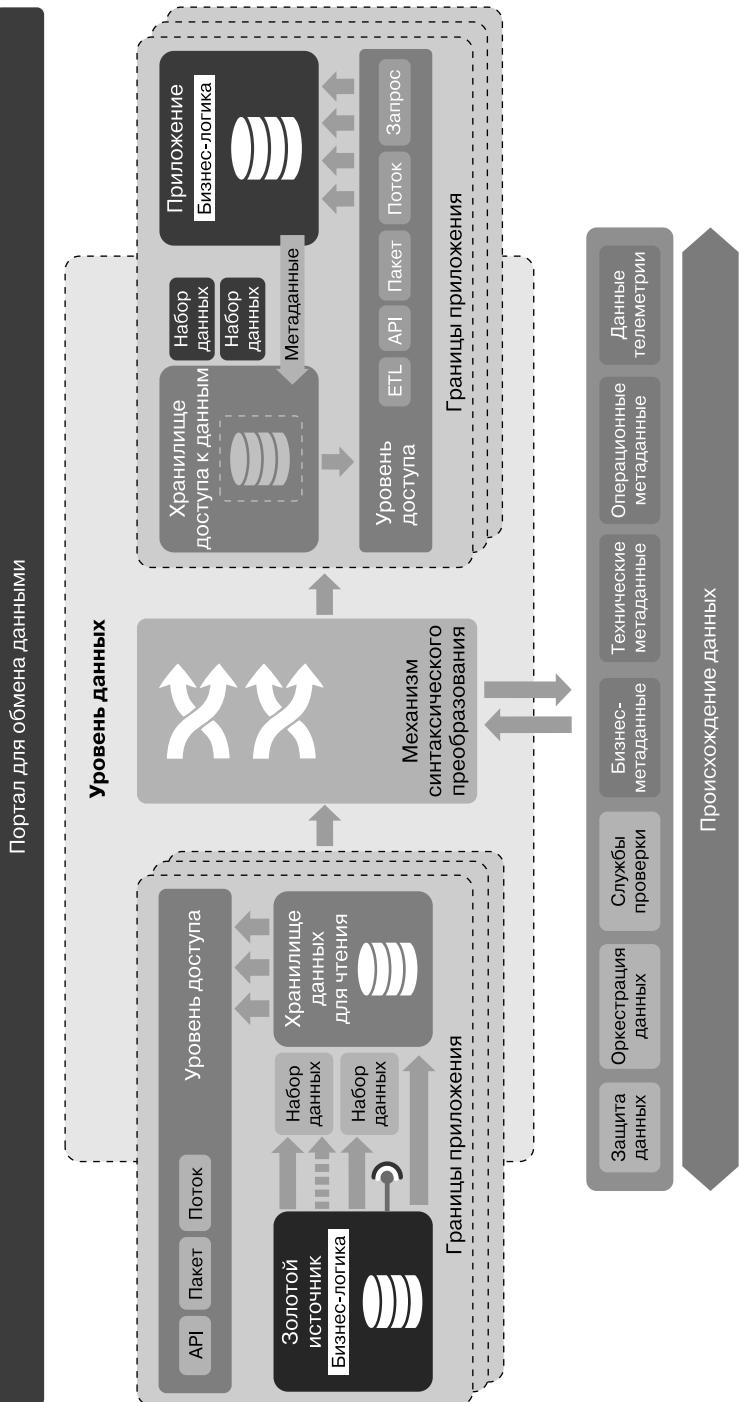


Рис. 3.16. Общая схема архитектуры, показывающая роль RDS в более широком плане

Данные можно получать из хранилища доступа по-разному. Например, можно предварительно обработать исторические данные. Данные можно запрашивать напрямую, получать через оптимизированный для чтения API, использовать потоки, читать как файлы и т. д. В следующих главах вы увидите, как разные компоненты интеграции могут работать вместе, упрощая разные архитектурные шаблоны.

Разумным шагом на данном этапе будет наполнение этого интеллектуального механизма большим количеством метаданных. Бизнес-метаданные и соглашения с другими потребителями могли бы улучшить способность механизма рекомендовать данные, нужные потребителям. Технические метаданные могут улучшить обработку данных за счет автоматического сбора и объединения нужных наборов, а метаданные безопасности и оперативные метаданные обеспечивают сохранность данных и сделают потребление более масштабируемым. Чтобы сделать данные более согласованными, можно расширить эту архитектуру с помощью решений для управления основными данными. Вы, скорее всего, заметили, как важны метаданные и совместная работа всех компонентов. Мы вернемся к этой теме в главах 6 и 10, с конкретными примерами, схемами и конструкциями.

Заполнение RDS по запросу

Проектирование облачных RDS отличается от проектирования локальных хранилищ, так как ресурсы хранения и вычислительные ресурсы разделяются. В облаке можно выбрать подход, позволяющий оставить полученные данные в файлах и папках входящего уровня, но заполнить оптимизированные для потребителей хранилища доступа к данным, расположенные близко к точке потребления по запросу. Эти экземпляры заполняются на основе соглашений о совместном использовании, хранящихся в центральном репозитории метаданных (рис. 3.17).

Облачные реализации можно легко расширить, добавив *каталог данных* и службу, позволяющие потребителям запрашивать разрешения¹. Благодаря этому можно относительно быстро ввести требования к самообслуживанию и площадке обмена данными, что мы уже обсуждали в предыдущем разделе. Вам может понадобиться итерационно добавить новые шаблоны RDS, такие как реляционные базы данных и хранилища документов, для более конкретного потребления данных. Обеспечивая надежное управление данными (возможность отчетности для поставщиков данных) и внедряя контракты на поставку данных

¹ Каталог данных – это хранилище метаданных в сочетании с инструментами управления данными и поиска, которое помогает пользователям находить данные и описывать их.

и службы проверки, вы избегаете тесной связи. С помощью метаданных вы сможете отслеживать использование и прозрачно показывать, какие поставщики и потребители данных подключены.

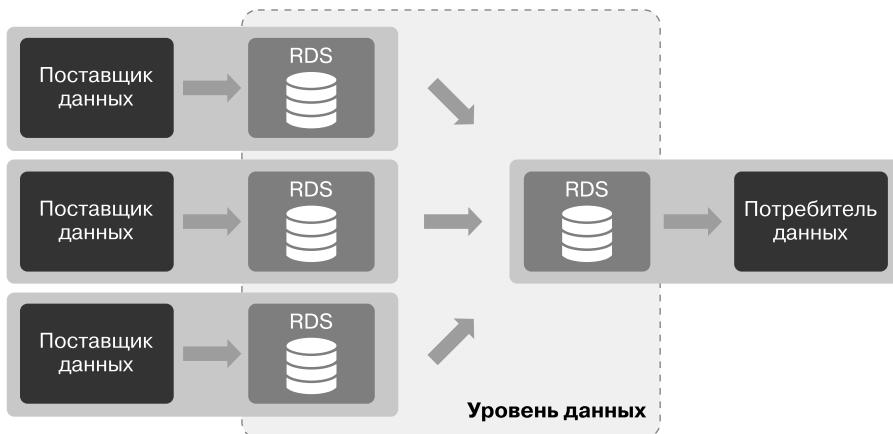


Рис. 3.17. В современной облачной самообслуживающейся среде экземпляры RDS могут быть подготовлены с использованием модели обслуживания по запросу. Вместо извлечения данные загружаются (копируются) и доставляются в место, близкое к их потребителю. С использованием функций модель может быть изменена на автоматическое потребление по мере доставки данных

Рекомендации по использованию RDS

Сложность при проектировании заключается в том, что RDS могут действовать как прямые источники для областей-потребителей или, если возникнет такая необходимость, области могут извлекать и дублировать их. Я говорю здесь о постоянных преобразованиях данных, которые относятся к процессу записи и хранения в новом месте (БД) за пределами архитектуры RDS. Избегайте создания неконтролируемых копий данных, хотя хотя в некоторых ситуациях предпочтительнее иметь копию поблизости или внутри приложения-потребителя. Мы обсудим это более подробно в главе 8.

Итак, общие рекомендации таковы.

- Данные RDS используются для создания новых данных. В целом это новые факты. Логически эти данные должны сохраняться в новом месте и менять владельца.
- Если нужно уменьшить общую задержку, порой лучше продублировать данные, чтобы они были доступны ближе к приложению-потребителю. Мы

подробнее рассмотрим это решение в главе 5, где вы узнаете, как создавать полностью контролируемые, распределенные, материализованные представления.

- Прямые преобразования данных и преобразования в масштабе реального времени бывают настолько тяжелыми, что могут ухудшить пользовательский опыт. Например, очень долгие запросы к RDS могут сильно разочаровать пользователей. Но все это можно предотвратить путем извлечения, реструктуризации и предварительной обработки данных или использования более быстрой БД.
- Если приложение-потребитель играет особенно важную роль, то иногда стоит дублировать данные и предотвратить отрицательные последствия от сбоя RDS.

Во всех вышеперечисленных случаях я рекомендую реализовать форму управления архитектурой предприятия, сосредоточенную на использовании RDS или создании новых хранилищ данных для предметных областей. Дублирование данных усложняет их очистку и получение представления об их использовании. Если данные можно копировать без ограничений или дополнительных затрат на управление, то их можно распространить дальше. Здесь нужно учесть еще одну потенциальную проблему — соблюдение таких правил, как GDPR или CCPA.

Итоги главы

RDS предназначены для передачи потребителям огромных объемов данных. Их не следует рассматривать как еще одно озеро или хранилище данных. RDS расширяют существующие приложения, наследуя их контекст. Они строго регулируются и остаются сильно изолированными. Это важно, потому что чем меньше количество общих связей, тем меньше усилий придется прилагать для координации команд.

RDS также используются для изоляции транзакционных и операционных систем и позволяют им обслуживать большие объемы данных. Они играют важную роль в управлении жизненным циклом данных. Поскольку контекст предметной области сохраняется, они являются идеальным кандидатом для оперативной аналитики. Области-поставщики часто замечают эти преимущества и сами начинают использовать RDS.

Еще в этой главе мы рассмотрели общую инфраструктуру и возможности RDS. Теоретически RDS могут быть развернуты и размещены в самих предметных областях, однако создание общих платформ делает среду более полезной и упрощает контроль политик. Я рекомендую начинать с центра и позволять всему остальному развиваться в процессе реализации.

Еще одна рекомендация — совместно разработать и реализовать архитектуру и ее шаблоны в соответствии с потребностями организации. Предоставьте группам, управляющим предметными областями, общие макеты. Не позволяйте одной команде создавать всю архитектуру, но дайте другим командам внести свой вклад или временно присоединиться к командам платформы. Если, например, потребуется организовать синхронизацию между RDS или если команде понадобится преобразовать реляционную схему в схему типа «ключ — значение», определите необходимое в виде шаблона или компонента многоократного использования. То же относится к интеграции и потреблению данных. При масштабировании и создании современной платформы данных требуется множество дополнительных самообслуживающихся компонентов приложений. В большинстве случаев они будут включать централизованно управляемые фреймворки ETL, платформы выборки изменившихся данных и инструменты для потребления данных в реальном времени.

В следующей главе мы рассмотрим архитектуру API, которая применяется для обмена данными между сервисами и распределения небольших объемов данных для сценариев использования в реальном времени и с низкой задержкой.

ГЛАВА 4

Управление сервисами и API: архитектура API

В этой главе будут описаны архитектура API, сервис-ориентированная архитектура (service-oriented architecture, SOA) и многие современные шаблоны данных. Мы рассмотрим интеграцию корпоративных приложений, проектирование сервисов и управление ими, модели сервисов (данных), микросервисы, сервисные сетки, GraphQL и многое другое. К концу этой главы вы получите полное представление о современных сервис-ориентированных архитектурах, используемых для поддержки масштабируемости в распределенных программных системах и системах реального времени. Вы поймете, как такая архитектура вписывается в общую картину и как она связана с архитектурой RDS, которую мы обсудили в главе 3.

Знакомство с архитектурой API

Архитектура API получила свое название от аббревиатуры API (application programming interface — прикладной программный интерфейс). API позволяет приложениям, программным компонентам и службам взаимодействовать напрямую. Операционные, транзакционные и аналитические системы, которым необходимо получать данные друг от друга или запускать процессы в режиме реального времени, — хорошие примеры шаблонов API. Сюда относится и взаимодействие с облачными компаниями, внешними компаниями и социальными сетями. Еще сервисы могут пригодиться в случаях, когда вы как компания хотите предложить широкие возможности взаимодействия с пользователями через свои цифровые онлайн-каналы. Например, представьте, что пользователи, посетившие веб-сайт туристического агентства, могут одновременно видеть доступные рейсы и расписания от оператора авиакомпании.

В большинстве случаев обмен данными через API выполняется синхронно, с использованием *шаблона «запрос/ответ»*, когда сервис отправляет запрос,

запрашиваемая система принимает и обрабатывает его, возвращая ответное сообщение. Объем передаваемых данных обычно невелик, что делает возможным обмен данными в режиме реального времени, что необходимо в случаях, когда важна малая задержка. Противоположный шаблон — *асинхронная передача данных*, когда не требуется немедленный ответ для продолжения обработки. Передаваемые данные могут доставляться с разными несинхронизированными или прерывистыми интервалами. Сообщения, отправленные посредством HTTP-вызовов API, могут помещаться в очереди сообщений для дальнейшей обработки. Эта модель подписки известна как *публикация и подписка* или сокращенно pub/sub (publish and subscribe).

Поскольку API также могут использоваться асинхронно и в сочетании с очередями сообщений, а также с событийно-ориентированной обработкой данных, основное внимание в этой главе я уделю синхронным взаимодействиям и обмену данными по схеме «запрос/ответ». В главе 5 я представлю событийно-ориентированную обработку, организацию очередей, асинхронную связь и покажу, как они пересекаются с SOA и API.

Архитектура API тесно связана с сервис-ориентированной архитектурой (SOA), появившейся более десяти лет назад. SOA широко используется и рассматривается как современный подход к разработке программного обеспечения и подключению приложений. Хотя некоторые архитекторы и инженеры говорят, что «SOA мертв», она активно используется и более чем актуальна благодаря таким тенденциям, как микросервисы, открытые API и SaaS.

Что такое сервис-ориентированная архитектура

Когда вы начнете читать о SOA, вы встретите много разных мнений. Автор и разработчик Мартин Фаулер (Martin Fowler) считает невозможным описать SOA (<https://oreil.ly/cAN9H>), потому что это понятие имеет разные смыслы для разных людей.

Для меня SOA — это связь между приложениями через веб-службы. Речь идет об экспорте *бизнес-функций*. Некоторые люди называют их *сервисами*, другие — *API*. SOA используется для абстрагирования и сокрытия сложности приложений, ведь внутреннее строение SOA приложения (и их сложность) уходят на второй план. Вместо этого предоставляется бизнес-функционал.

SOA стандартизировала способ взаимодействия приложений. В рамках этой архитектуры обмен данными между приложениями осуществляется через стандартные протоколы, такие как SOAP или JSON, и общие архитектурные стили программного обеспечения, такие как REST. SOA — это синхронная (ожидание ответа для продолжения) и асинхронная связь (отсутствие ожидания).

Но и это не все. SOA разделяет и устанавливает четкие границы между приложениями и предметными областями. Это позволяет приложениям изменяться независимо, без необходимости изменять другие приложения. Таким образом, SOA можно использовать как стратегию быстрой разработки и поддержки приложений со сложными ландшафтами.

РЕСУРСЫ И ОПЕРАЦИИ

REST — это архитектурный стиль, который определяет набор ограничений и принципов абстракции, используемых для создания веб-сервисов¹. Ключевым понятием в REST являются *ресурсы* (<https://oreil.ly/lk2oo>). Под ресурсами подразумевается все, что достаточно важно, чтобы абстрагировать их и ссылаться на них как на внутренние компоненты. Ресурсом может быть любой объект или источник информации, который можно однозначно идентифицировать, — например, клиент, контракт, учетная запись, заказ или продукт. Для API на основе REST протокол HTTP имеет набор операций, определяющих взаимодействия с такими ресурсами.

- *POST* — метод для *создания* нового ресурса.
- *GET* — метод для *чтения* одного или нескольких (или даже всех) ресурсов.
- *PUT* — метод для *обновления* или замены любого существующего ресурса. Для обновления только определенных полей в ресурсе чаще используется метод *PATCH*.
- *DELETE* — метод для *удаления* определенного ресурса.

Эти элементарные операции часто сочетаются с методом CRUD (create, read, update, delete — «создание, чтение, обновление, удаление»), где REST — это стиль протокола API, а CRUD — стиль взаимодействия для управления данными.

Некоторые архитекторы и инженеры утверждают, что SOA — это средство доступа к бизнес-функциям или процессам, а не только к данным. В определенном смысле это действительно так, но важно понимать, что основной элемент всегда — данные. Если коммуникационные и сетевые пакеты не передают никаких данных, то связь никогда не будет установлена и SOA перестанет работать. Поэтому SOA во многом зависит от данных. Другие архитекторы и инженеры говорят, что она используется только в сфере операционных и транзакционных систем. Я не считаю это полностью верным, потому что SOA — это архитектура для взаимодействий приложений как из операционного, так и из аналитического мира в реальном времени.

Для меня самым большим прорывом в SOA является абстракция. Она заключается в том, что функционал и данные инкапсулируются сервисами. Приложения

¹ Рой Филдинг (Roy Fielding) дал определение REST в своей докторской диссертации *Architectural Styles and the Design of Networkbased Software Architectures* в 2000 году (<https://oreil.ly/J7B1r>).

вместо прямых взаимодействий друг с другом обмениваются данными через четко определенные сервисные интерфейсы (рис. 4.1) с использованием стандартных шаблонов взаимодействия. Сложность и данные внутри приложения, реализующего сервис, остаются скрытыми и больше не беспокоят потребителей¹.

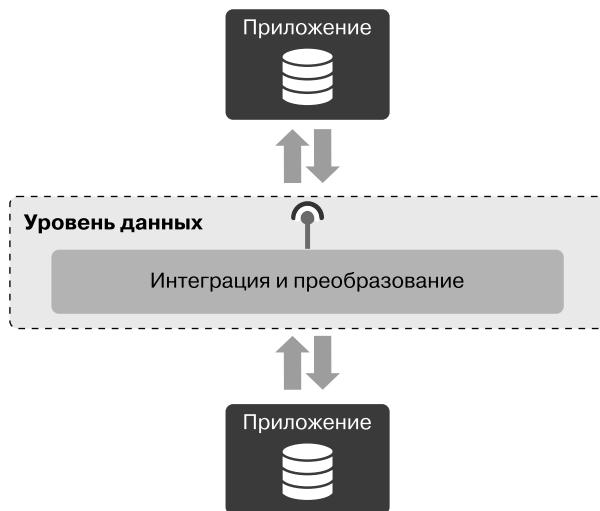


Рис. 4.1. Вместо прямого вызова приложения обмен данными в SOA осуществляется через вызовы API. API обычно имеет другую структуру и скрывает внутреннюю сложность приложения. Таким образом данные преобразуются

ТЕРМИНОЛОГИЯ SOA

Роли поставщика и потребителя услуг в SOA аналогичны ролям поставщика и потребителя данных, которые использовались в предыдущих главах.

- *Поставщик услуг* — это приложение, реализующее сервис для предоставления функциональных возможностей и данных.
- *Потребитель услуг* — это приложение, использующее сервис и которому передаются запрошенные данные.

Для лучшего понимания в этой главе я буду использовать термины «поставщик услуг» и «потребитель услуг», но в следующих главах я вернусь к «поставщику данных» и «потребителю данных».

¹ Наблюдение за инкапсуляцией и внутренней сложностью приложения и потоком данных между приложениями через службы было очень хорошо описано Пат Хелланд (Pat Helland) в статье Data on the Outside Versus Data on the Inside (<https://oreilly.com/online6gr>).

История SOA началась в 2009 году, когда в Open Group описали ее в официальном документе, который в конечном итоге превратился в *SOA Source Book*. Open Group определяет SOA как «архитектурный стиль, поддерживающий сервис-ориентированность. Ориентация на сервис — это образ мышления с точки зрения предоставления услуг, разработки на основе услуг, а также их результатов».

Первоначальной целью SOA было сделать ИТ-ландшафт более гибким и обеспечить связь между приложениями в реальном времени. До SOA большая часть взаимодействий с корпоративными приложениями осуществлялась через прямые вызовы «клиент — сервер» (вызовы удаленных процедур (remote procedure call, RPC))¹.

Недостатком прямого взаимодействия приложений является связь с базовой средой выполнения приложения. Для обмена данными между приложениями, например, с помощью методов RPC нужны одни и те же клиент-серверные библиотеки и протоколы. Это не позволяет использовать большое разнообразие приложений, так как библиотеки и протоколы доступны не для всех языков. Еще один недостаток — совместимость. Одновременно поддерживать разные версии сложно. Если метод, экспортруемый приложением, изменится, то другое приложение тоже придется изменить, иначе взаимодействие может стать невозможным. Последний недостаток — масштабируемость: с увеличением количества приложений количество интерфейсов начинает расти неуправляемо. Обычно те же проблемы возникают при построении двухточечных интерфейсов между приложениями.

Эти недостатки являются основными причинами, почему специалисты начали работать над новой архитектурой, основанной на абстракции более высокого уровня и ключевых концепциях интерфейсов приложений, сервисов, репозиториев сервисов и сервисных шин. Приложения и специфика их протоколов в этой модели скрыты, а функциональные возможности и данные экспортируются через сервисы, использующие стандартные веб-протоколы, такие как HTTP. Связь и интеграция между приложениями осуществляется через сервисные шины. Внедрение технологий и методологий для поддержки этой формы связи между (корпоративными) приложениями известно под названием «интеграция корпоративных приложений»².

¹ RPC похож на модель «клиент — сервер». Это шаблон, позволяющий одной программе использовать и запрашивать функциональные возможности у другой программы, находящейся в иной системе в сети. Для связи между клиентом и сервером вместо сообщений в RPC используется язык описания интерфейса (interface description language, IDL). Интерпретатор IDL должен быть реализован с обеих сторон.

² Интеграция корпоративных приложений (enterprise application integration, EAI) — это использование технологий и услуг на предприятии для интеграции программных приложений и аппаратных систем.

Интеграция корпоративных приложений

В начале 2000-х годов, когда интернет стал более популярным, появилась *корпоративная сервисная шина* (enterprise service bus, ESB) — новая платформа интеграции корпоративных приложений, часть SOA, предназначенная в том числе для управления обменом данными между различными приложениями. Назначение сервисной шины — соединение всех участников сервисов и приложений друг с другом. В нее добавлены новые функции, помимо традиционных для промежуточного ПО обработки сообщений (message-oriented middleware, MOM), и клиент-серверных моделей¹.

- *Оркестрация сервисов* — объединение нескольких детализированных сервисов в один составной сервис более высокого порядка.
- *Маршрутизация на основе правил* — получение и рассылка сообщений на определенных условиях.
- *Преобразование сообщений в стандартные форматы* — преобразование из одного конкретного формата данных в другой (включая преобразование формата обмена данными, например SOAP/XML в JSON).
- *Гибкое посредничество* — поддержка нескольких версий одного и того же интерфейса для совместимости.
- *Адаптация к подключению* — поддержка широкого спектра систем для связи с серверным приложением и преобразования данных из формата приложения в формат шины.

ESB важна для маневренности и гибкости, она позволяет компаниям быстрее доставлять услуги с использованием шин и адаптеров. ESB превращает традиционные приложения в современные сервисы с простыми интерфейсами. ESB также разделяет приложения, предоставляет абстракции и обеспечивает посредничество. Адаптер соединяет приложения и ESB, а также выполняет преобразование протоколов между ними. Принцип работы ESB показан на рис. 4.2.

Чтобы упростить реализацию ESB, многие поставщики программного обеспечения готовили свои приложения и платформы для SOA, модернизируя их. Интерфейсы сервисов во многом основаны на стандартах Консорциума Всемирной паутины (W3C), таких как протокол простого доступа к объектам (Simple Object Access Protocol, SOAP) с расширяемым языком разметки (eXtensible Markup Language, XML) в качестве формата обмена сообщениями по умолчанию.

¹ Промежуточное ПО обработки сообщений (<https://oreil.ly/Bli92>) — это программная или аппаратная инфраструктура, которая поддерживает отправку и получение сообщений между分散ными системами; ключевой строительный блок ESB, используемый для маршрутизации сообщений и их надежной обработки.

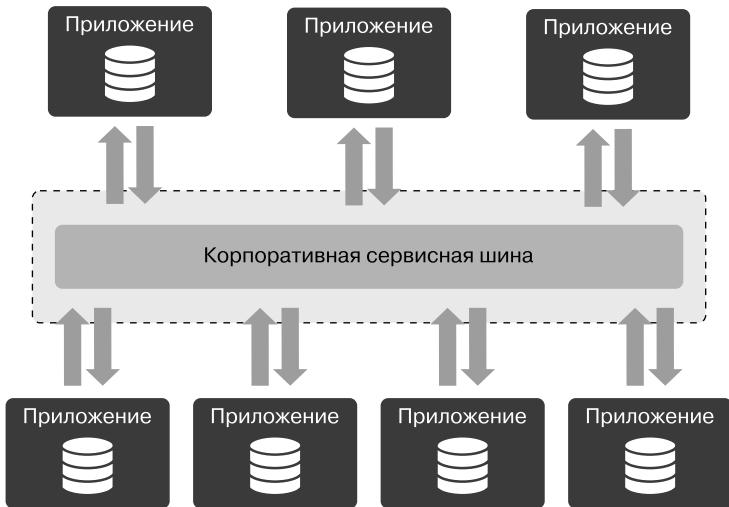


Рис. 4.2. Вместо того чтобы напрямую взаимодействовать друг с другом, приложения взаимодействуют через ESB (корпоративную сервисную шину)

ЧТО ТАКОЕ WSDL, XSD И СЕРВИСНЫЕ РЕПОЗИТОРИИ

Связь на основе XML обычно поддерживается с помощью WSDL, XSD и сервисных репозиториев. Язык описания веб-сервисов (Web Services Description Language, WSDL), также написанный на XML, является стандартом описания веб-сервисов и их операций. Описание на этом языке определяет методы, местоположение, типы данных и транспортный протокол. Это упрощает работу по интеграции и разработке программного обеспечения, поскольку разработчик может полностью создать законченный веб-сервис, просто предоставив описание на WSDL.

WSDL-документы связаны с другими документами на основе XML: XSD (XML Schema Definition — определение схемы XML). Документы XSD описывают схему, определяющую структуру XML-документов. Они позволяют убедиться, что XML-документ действителен и соответствует правилам, изложенным в схеме.

XML-документы обычно хранятся в сервисном репозитории. Как правило, он представляет собой реестр на основе XML, в котором публикуются доступные веб-сервисы. Репозиторий также отслеживает версии, документацию и политики.

Популярность SOAP обусловлена использованием протокола передачи гипертекста (Hypertext Transfer Protocol, HTTP) для сетевых взаимодействий и строгой структурой, аналогичной языку разметки (HyperText Markup Language, HTML), который используется для создания веб-сайтов. Те же стандарты HTML и HTTP были и остаются основой современной интернет-связи. Базируясь на этих стандартах, ESB действует как посредник по преобразованию и интеграции, обеспечивая универсальную связь между поставщиками и потребителями услуг.

Помимо преобразования форматов сообщений и придания приложениям современной коммуникационной оболочки, ESB также предоставляет разные варианты коммуникации в случаях, когда требуется более сложное взаимодействие. Два из них — это оркестрация и хореография сервисов. Они будут подробно рассмотрены в следующих разделах, потому что именно они делают SOA более сложной и трудоемкой.

Оркестрация сервисов

В SOA роли поставщика и потребителя услуг иногда не видны при объединении нескольких сервисов. Они могут быть реализованы разными способами, что еще больше усложняет интеграцию и потребление данных, поэтому давайте рассмотрим некоторые варианты и их различия.



ESB внутри SOA могут стать сложными монолитами. Я также упомяну об этом в подразделе «Сходства между SOA и архитектурой корпоративного хранилища данных» на с. 128, сравнивая SOA с корпоративным хранилищем данных.

Первый шаблон — *агрегирование сервисов*, когда несколько сервисов объединяются, переплетаются между собой и представляются как единый, более крупный агрегированный сервис. Это может быть полезно для избавления от сложностей при управлении и удовлетворении нужд (одного или нескольких) потребителей услуг. Агрегатор в этом сценарии отвечает за координацию, вызов и объединение разных сервисов.

В рамках агрегирования также определяется *оркестрация сервисов*, координация и выполнение задач для объединения и интеграции различных сервисов¹. Она может быть кратковременной, с использованием довольно простых комбинаций, как в шаблоне агрегирования, или длительной, с объединением сервисов в более сложный поток, что требует реализации логики принятия решений и нескольких путей выполнения. Несмотря на то что оркестрация сервисов не дает точных данных о продолжительности работы, люди используют этот термин преимущественно в контексте сервисов, работающих относительно долгое время. Термин «оркестрация сервисов» страдает некоторой неоднозначностью, потому что может использоваться по разным причинам.

¹ Многие еще используют термин «композиция сервисов». Принципы композиции сервисов поощряют проектирование сервисов таким образом, чтобы их можно было легко повторно использовать в нескольких решениях, а это означает, что существующие сервисы можно использовать для создания новых. Хотя композиция обозначает тот факт, что сервисы объединены, она не подразумевает способ объединения.

Типичные причины оркестрации — технические. Они обычно видны и появляются в контексте составных сервисов при объединении, разделении, преобразовании, проверке и удалении данных. В некоторых случаях требуется простой поток услуг между сервисами¹. Например, чтобы открыть контракт с клиентом, сначала необходимо вызвать сервис продуктов другого клиента.

ОРКЕСТРАЦИЯ С ИСПОЛЬЗОВАНИЕМ ESB ИЛИ С ТОЧКИ ЗРЕНИЯ ПОТРЕБЛЕНИЯ?

Есть два подхода к объединению и интеграции сервисов.

Первый — использовать службы агрегирования ESB, которые позволяют потребителям получать все данные одним вызовом. ESB играет роль оркестратора и выполняет всю связанную этим тяжелую работу. ESB в этом сценарии, становится единственным источником технических проблем.

Альтернативный подход — позволить потребляющим приложениям играть роль оркестратора. Естественным следствием этого является то, что приложение должно управлять порядком вызовов и объединять (интегрировать) результаты.

У обоих подходов есть свои плюсы и минусы. Преимущество управления оркестрацией в ESB заключается в возможности многократного использования и упрощении потребления данных. Недостаток — ESB должна обеспечивать адекватную производительность при объединении всей логики. Более серьезный минус — роль поставщика услуг становится не такой выраженной. Так происходит из-за того, что агрегирование услуг действует как фасад для различных сервисов. Агрегированные сервисы обычно объединяют сервисы от разных владельцев. В таких случаях ни один владелец не может полностью гарантировать целостность службы агрегирования.

Оркестрация может выполняться и по другой причине: когда необходимо управление бизнес-процессами (business process management, BPM). Процессы в этой ситуации содержат бизнес-логику и обычно выполняются долго. Координация, необходимая для управления последовательностью шагов, тоже требует, чтобы статусы процесса (состояние каждого шага) сохранялись и находились в базе данных. Долгосрочные бизнес-процессы могут прерываться и ждать выполнения других процессов (или задач пользователя). Как правило, они асинхронны, поэтому для процессов таких типов шаблон оркестрации сервисов обычно сочетается с визуальным и интуитивно понятным программным обеспечением BPM для управления сложными и длительными процессами. Примеры таких продуктов — BPM от TIBCO (<https://oreil.ly/3y0gI>) и Camunda (<https://camunda.com/>).

¹ Потоки услуг, также называемые микропотоками, не прерываются и обычно действуют в одном потоке выполнения и только в одной транзакции. Они непродолжительны и, как правило, используют только синхронные службы. Состояние процесса обычно не сохраняется.



На уровне процессов программного обеспечения BPM содержитя вся информация о потоке обмена данными между участниками. Эта информация включает последовательность, зависимости, статус, данные об участнике, параллельные задачи, правила ведения бизнеса и т. д. *Никогда* не следует хранить данные приложения на уровне процесса: это распространенная ошибка. Вместо этого используйте BPM API для запуска процессов и получения информации о состоянии рабочего потока. Не используйте его для хранения данных о клиентах, продуктах или контрактах, потому что он не предназначен для этой цели.

BPM обычно используется, если вам нужно организовать (длительные) процессы в разных приложениях из разных областей. Это помогает областям отделить свои бизнес-процессы от логики бизнес-приложений. Однако при организации процессов и данных в границах приложения лучше делать это индивидуально, внутри самого приложения.

Когда BPM управляет длительными процессами, обработка обычно выполняется асинхронно. BPM руководит всеми зависимостями, отслеживает индивидуальное состояние каждого процесса и знает, когда запускать, извлекать и сохранять данные. Поэтому API, которые используются для запуска или вызова процессов, иногда называют *API процессов*.

Оркестрацию сервисов часто путают с другими типами оркестрации. В контексте сервис-ориентированной архитектуры оркестрация — это вызов и выполнение операционных и функциональных процессов, необходимых для предоставления сквозного обслуживания. Обычно эта форма оркестрации использует прикладные API и может управляться программным обеспечением BPM. Она отмечена значком ❸ на рис. 4.3.

Другие формы оркестрации, в том числе планирование, автоматическая настройка и координация, включают:

- планирование процессов в операционных системах ❶, например, с помощью cron в Unix-подобных системах;
- планирование процессов и задач в границах приложения ❷, например, с использованием внутренних/собственных планировщиков приложений;
- оркестрацию данных ❸ для автоматизации обработки данных, например перемещения или подготовки данных и процессов ETL. Apache Airflow — распространенный инструмент для такого рода действий;
- планирование и оркестрацию инфраструктуры ❹, например оркестрацию виртуальных машин, выключение систем и т. д. Популярные инструменты в этой области — Ansible (<https://www.ansible.com/>), Puppet (<https://puppet.com/>), Salt (<https://www.saltstack.com/>) и Terraform (<https://www.terraform.io/>).

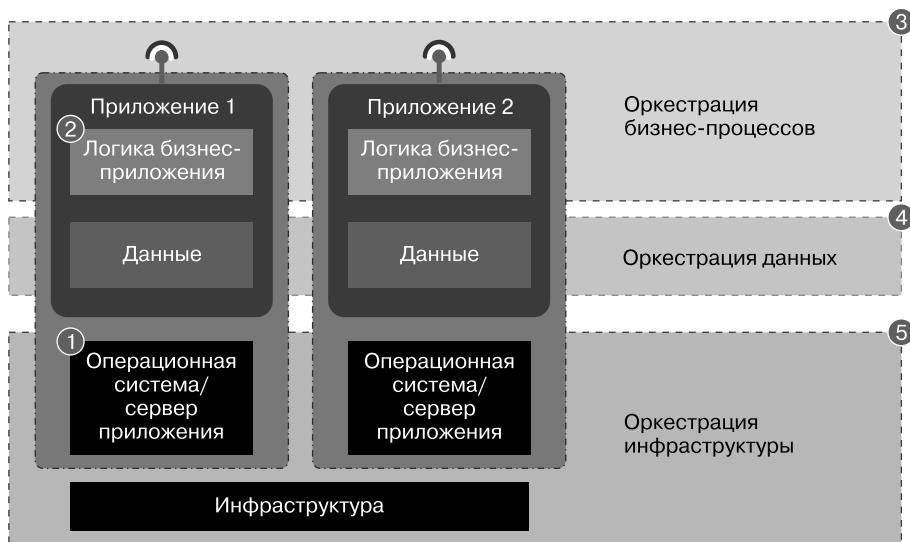


Рис. 4.3. Оркестрация — это общий термин для автоматизации рабочего процесса.
Оркестрацию можно использовать в самых разных контекстах

Дополнительной формой оркестрации является непрерывная интеграция и непрерывная доставка (continuous integration/continuous delivery, CI/CD), то есть автоматизированный процесс мониторинга, сборки, тестирования, выпуска и интеграции программного обеспечения. Эта форма оркестрации обычно использует и сочетает в себе многие другие ее формы, поэтому на рис. 4.3 она не показана.

Хореография сервисов

Другой шаблон, который скрывает роли поставщиков и потребителей услуг, — это *хореография сервисов*, которая относится к глобальному распределению взаимодействия процессов и бизнес-логики между независимыми сторонами, сотрудничающим для достижения определенной бизнес-цели. С помощью хореографии сервисов логика принятия решений становится распределенной: каждая служба наблюдает за окружением и действует автономно. Не существует централизованной точки, как в случае с BPM, и ни одна из сторон не имеет полного контроля над процессами других. Каждая сторона владеет частью общего процесса. В рамках хореографии сервисов, как можно видеть на рис. 4.4, запросы передаются туда и обратно, создавая эффект пинг-понга между различными поставщиками услуг и их потребителями.

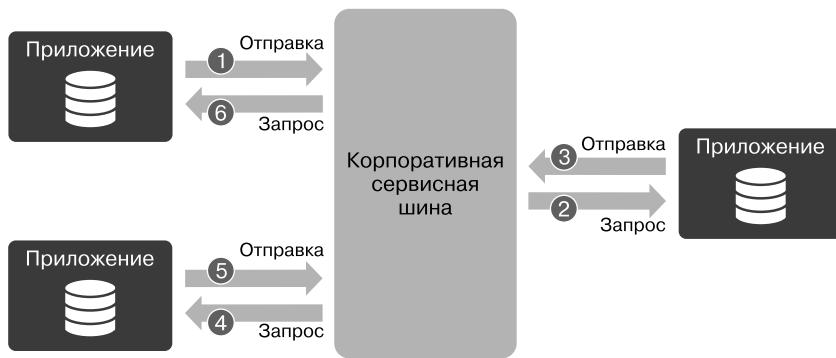


Рис. 4.4. Хореография сервисов — децентрализованный подход к использованию сервисов и организации взаимодействий между ними. В этом примере логика выбора сервисов, которые должны вызываться дальше, распределена между участниками. Каждому принадлежит определенная часть процесса. Логика, управляющая взаимодействием между сервисами, находится за пределами центральной платформы. Сервисы сами решают, что вызвать дальше и в каком порядке

Можно также сочетать хореографию, агрегирование и оркестрацию сервисов. Некоторыми сервисами и процессами можно управлять централизованно, тогда как другие могут независимо запускать рабочие процессы из других агрегированных сервисов. Во всех ситуациях роли поставщика услуг и потребителя скрыты, потому что предметные области вряд ли могут гарантировать полную целостность.



Мартин Фаулер предложил интересный шаблон под названием «Сотрудничество по событиям» (<https://oreil.ly/ajRx->). Он нужен для приложений, которые обмениваются данными и зависят от информации из других приложений. Вместо того чтобы делать запросы, когда требуются данные, приложения генерируют события, когда что-то меняется. Другие приложения получают эти события и реагируют соответствующим образом. Этот шаблон хорошо подходит для архитектуры потоковой передачи, которую мы обсудим в главе 5.

В двух предыдущих разделах я в основном сосредоточился на рассмотрении сервисов через взаимодействие процессов и агрегирование. В следующем разделе я хочу более подробно изучить другой способ различения сервисов, через наблюдение за контекстом, в котором они работают.

Публичные и частные сервисы

Временами люди делят сервисы SOA на два типа: публичные и частные. *Публичные сервисы*, иногда называемые *бизнес-сервисами*, предоставляют определенные бизнес-функции. Они важны для конкретной бизнес-задачи и способствуют ее

решению. Одни предоставляют сгруппированные данные и делают это так, как нужно бизнесу. Другие сервисы инициируют обработку, запускают рабочие процессы или затрагивают поведенческие аспекты. Бизнес-сервисы, как правило, абстрактны, являются основой для конкретных бизнес-функций и работают вне границ приложений.

Частные или технические сервисы — это сервисы, которые не обязательно отражают бизнес-функции. Они используются как часть внутренней логики и предоставляют технические данные или функциональные возможности инфраструктуры, такие как отображение таблицы базы данных. Частные сервисы могут входить в состав публичных. В этом случае они обертываются другими сервисами, которые в конечном итоге обеспечивают бизнес-функционал.



Некоторые используют термины «сервисы инфраструктуры» или «сервисы платформы». Это сервисы, абстрагирующие инфраструктуру от предметных областей. Они обычно стандартизированы в своей организации и включают функции, которые применяются ко всем областям, такие как аутентификация, авторизация, мониторинг, журналирование, настройка или аудит.

Частные сервисы имеют более высокую степень связности, потому что не абстрагируются от задач. Некоторые говорят, что эти сервисы не следует публиковать в центральном репозитории из-за того, что они не подходят для широкой аудитории и работают только в пределах приложений.

Модели сервисов и канонические модели данных

При подключении разных приложений инженерам и разработчикам необходимо общее понимание моделей данных и бизнес-функций, предоставляемых разными сервисами. Для этого нужны модели сервисов и канонические модели данных. *Модель сервиса* описывает, как выглядит интерфейс сервиса и как моделируются данные: их суть, отношения и атрибуты. В силу своего разнообразия она может включать определения, зависимости от других сервисов, номера версий, синтаксис, протокол и другую информацию, такую как владельцы приложений, состояние производства и т. д.

Цель модели сервиса — предоставить информацию, упрощающую использование и интеграцию сервисов. Обычно она не зависит от платформы и не пытается описать систему. Модели сервисов используются в разных сферах и существуют в разных форматах. Некоторые применяются только в документации и описывают наиболее часто используемый контекст, другие живут в инструментах в виде кода и описывают данные и все их атрибуты и протоколы.

Канонические модели данных отличаются от моделей сервисов тем, что они нацелены на определение сервисов и языков стандартным и унифицированным образом. Тогда как модели сервисов остаются ближе к приложению. Канонические модели используются для согласования и стандартизации различных моделей сервисов. Некоторые специалисты применяют унификацию ко всем сервисам, используя глобальную схему. В результате канонические модели обычно становятся большими и сложными.

Сходства между SOA и архитектурой корпоративного хранилища данных

ESB, агрегирование и многоуровневость атомарных сервисов, а также унификация с использованием центральной канонической модели делают SOA сложнее, чем нужно. Я провожу параллель между традиционными реализациями SOA и архитектурой корпоративных хранилищ данных, потому что многие из их задач схожи.

Размер канонической модели

Первое сходство — масштаб, в котором используются канонические модели. Многие организации пытаются использовать единую каноническую модель предприятия для описания *всех сервисов*. Каноническая модель предприятия требует перекрестной координации между всеми командами и вовлечеными сторонами. Сервисы в этом подходе обычно получают много дополнительных атрибутов из-за того, что каждая сторона настаивает на отражении своих уникальных требований. В результате получается монстроподобная модель интерфейса, пытающаяся учесть потребности всех и каждого, но в итоге никому не подходит. Она настолько усреднена, что никто не может сказать, какие именно услуги предоставляются.

ESB как оболочка для устаревшего промежуточного ПО

Следующее сходство — это разбиение на уровни и агрегирование сервисов. Прагматический подход к предоставлению новых услуг заключается в быстром превращении (рис. 4.5) существующих сервисов в новые. Прежде чем вы это осознаете, возникнет бесконечный каскад вызовов сервисов. Сервисы вызываются, не зная точно, откуда поступают данные или какие сервисы или процессы выполняются. Обычно то же самое происходит с BPM. Техническая оркестрация и оркестрация процессов смешаны, или вся логика предметной области и процесса объединяется централизованно, что затрудняет наблюдение за тем, какие процессы принадлежат друг другу.

Типичный уровень интеграции, который мы видели в старых системах промежуточного программного обеспечения, — еще одна интересная загадка для многих организаций. До ESB использовались дополнительные уровни и компоненты промежуточного ПО¹. Многие из этих архитектур до сих пор существуют и выполняют большую часть работы по маршрутизации сообщений, интеграции, преобразованию и оркестрации процессов, включая управление и сохранение состояния многих из этих процессов. Некоторые предприятия объединяют эти унаследованные системы промежуточного программного обеспечения с ESB. Это снижает сложность и открывает возможность использовать более современные виды взаимодействий, например JSON вместо RPC. Добавление старого промежуточного программного обеспечения в новую оболочку ESB увеличивает сложность, потому что продолжается использование многоуровневого выполнения задач (см. рис. 4.5). Различные платформы и системы накладываются друг на друга, поэтому каждое изменение в серверных системах должно выполняться в нескольких разных местах.

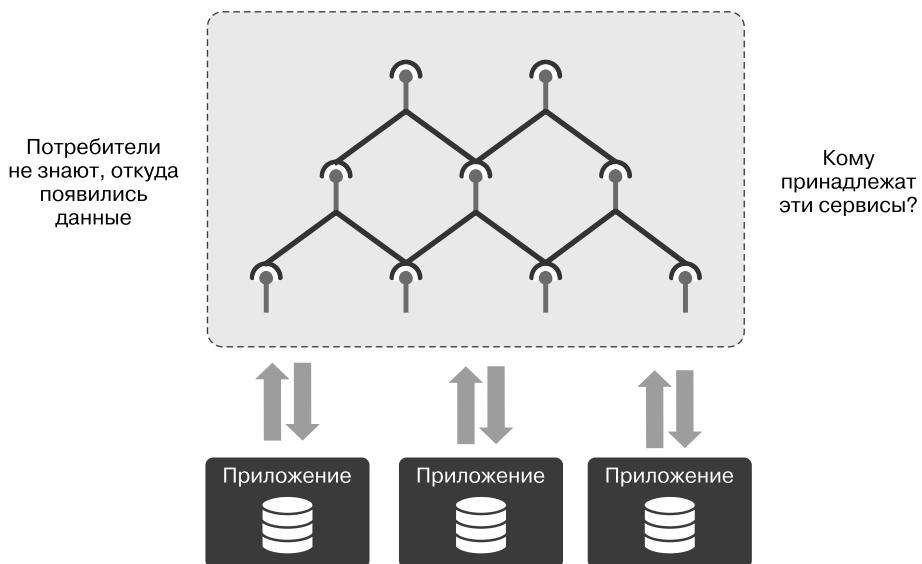


Рис. 4.5. Один из рисков наложения одних сервисов на другие заключается в том, что данные извлекаются, но никто не знает, откуда они пришли. Другой риск заключается в том, что никто не желает брать эти сервисы на себя

¹ Промежуточное ПО или промежуточное ПО, ориентированное на сообщения (message-oriented middleware, MOM), — это общий термин для технологий и продуктов, поддерживающих отправку и получение сообщений между системами. ESB можно рассматривать как промежуточное ПО, но особое внимание уделяется аспектам интеграции приложений и данных.

ESB-управление состоянием приложения

Последнее наблюдение, которым я хочу поделиться, касается позиционирования ESB для управления состоянием¹. Вместо того чтобы заботиться о состоянии в приложении или о возможности управления бизнес-процессами, ESB используется для оркестрации длительных бизнес-процессов или для сохранения состояния приложений. В результате ESB превращается в базу данных. Риск такого подхода — слишком сильная созависимость. Если ESB отключится по какой-либо причине, информация о состояниях будет потеряна и все процессы могут утратить свое фактическое состояние. Другой возможный риск связан с изменением приложений: если требуется изменение состояния и все приложения полагаются на состояние, хранящееся в ESB, потребуется тщательная перекрестная координация для точного внесения всех изменений. ESB, в принципе, не должна иметь состояния, за исключением временного управления сеансом и кэшированием.

Современный взгляд на SOA

Платформы интеграции ESB сделали SOA сложнее, чем следовало бы. Предприятия буквально восприняли букву E (Enterprise — «корпоративная») в ESB и внедрили в свои организации монолиты, чтобы позаботиться об интеграции всех сервисов. Центральные команды диктовали, как должны осуществляться многократное использование и проектирование сервисов, а интенсивное перекрестное сообщение мешало внедрению инноваций и гибкости команд.

Между тем взгляды на модульность приложений и архитектур изменились с появлением микросервисов. Благодаря цифровой экосистеме, облачным технологиям, SaaS и API-экономике API распространились за пределы предприятий. Современные базы данных и приложения поддерживают REST API по умолчанию. Неудивительно, что появилась необходимость в новом взгляде на SOA.

API-шлюз

Создание и интеграция сервисов долгое время были трудными задачами. Серверные приложения были сложными, и преобразование данных в правильный формат часто было проблемой. ESB давала компаниям большое преимущество, поскольку она легко отображала сложные приложения как современные, используя их возможности коммуникации и преобразования.

С появлением REST API и тенденций к модернизации приложений коммуникация и интеграция сервисов между приложениями начали меняться. Переда-

¹ Управление состоянием относится к управлению состоянием приложения — вводимыми пользователем данными, состояниям процессов и т. д.

ча репрезентативного состояния (Representational state transfer, REST) – это архитектурный шаблон, используемый в современных веб-приложениях для передачи информации при отсутствии состояния¹. Он основан на простоте: ресурсы – простые блоки связанной информации – идентифицируются с помощью универсальных идентификаторов ресурсов (Uniform Resource Identifiers, URI) и могут быть связаны с гипермейдийными ссылками². Он также использует HTTP с унифицированными методами интерфейса. REST API стали популярными благодаря своей функциональной совместимости и гибкости, и теперь они используются на веб-сайтах, в мобильных приложениях, играх и др.



REST API обычно используют CRUD-операции (create, read, update, delete – создание, чтение, обновление, удаление), которые отображаются в элементарные операции с базой данных или репозиторием. Как правило, записи или объекты данных обрабатываются напрямую. REST, с другой стороны, работает с представлениями ресурсов, каждое из которых идентифицируется URL-адресом. Обычно это не объекты данных, а сложные высокоуровневые абстракции. Например, ресурсом может быть контракт с заказчиком. Это означает, что ресурс – это не только запись в таблице «контрактов», но также отношения с ресурсом «пользователь», «соглашение», к которому прикреплен контракт, и, возможно, какое-то другое содержимое, которому он принадлежит.

Когда REST API стали популярными, форматы сообщений и протоколы также начали меняться. Протокол SOAP с его относительно многословным форматом сообщений XML заменил JSON (JavaScript Object Notation – нотация объектов JavaScript). JSON более быстрый, потому что имеет более простой синтаксис. Еще одно важное преимущество в том, что JSON-сообщения легко кэшируются или сохраняются в документно-ориентированных БД. Платформы API и интеграции могут хранить данные, которые были запрошены ранее, а затем обслуживать их по запросу.



В некоторых случаях положение REST API в общей иерархии может скрываться от потребителей за дополнительными уровнями или посредниками, в соответствии с инструкциями. Обычно это необходимо для увеличения масштабируемости (кэширование и балансировка нагрузки) и безопасности.

¹ Codecademy объясняет передачу репрезентативного состояния (Representational State Transfer, REST) (<https://oreil.ly/9RKWZ>).

² HATEOAS (Hypermedia as the Engine of Application State – гипермедиа как механизм состояния приложения) используется для связывания ресурсов REST. Существует руководство по API на основе REST (<https://oreil.ly/BnfAg>).

Современные приложения и изменения в протоколах и структурах сообщений тоже повлияли на сервисную шину предприятия. Появился более легкий компонент интеграции — *API-шлюз*. Он не обременен адаптерами или сложными интеграционными функциями ESB, но по-прежнему допускает инкапсуляцию и предоставляет возможности для контроля, защиты, управления и составления отчетов об использовании API.

Модель ответственности

Как должна выглядеть наша новая SOA, исходя из этих тенденций и необходимости разбить монолит ESB? Вместо использования центральной модели ответственность за создание и предоставление услуг будет возвращена базовым предметным областям. В децентрализованной модели все области могут развиваться со своей скоростью, не задерживая другие. В этой новой модели ответственность за бизнес-логику, выполнение процесса, сохранение данных и проверку контекста также возвращается областям. Использование шлюза ESB или API в качестве (централизованной) базы данных будет запрещено; агрегирование и оркестрация разрешены только внутри области. Эта модель изображена на рис. 4.6.

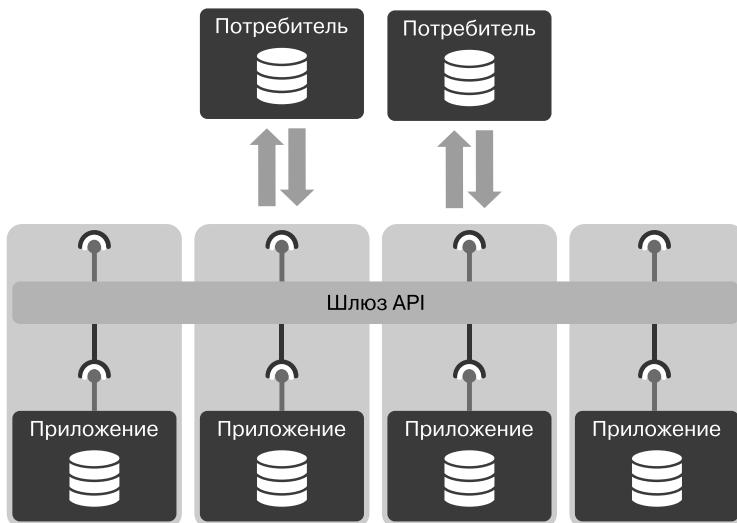


Рис. 4.6. Вместо использования тяжелой ESB службы и API будут отделены от областей через облегченный шлюз API

Как видите, такая модель полностью соответствует принципам проектирования, представленным в главе 2. Вместо того чтобы направлять все данные

и интеграцию из областей в центральную ESB, области должны сами разрабатывать, поддерживать и предоставлять свои услуги. Здесь будут применяться те же принципы, что и для архитектуры, оптимизированной для чтения.

Чтобы упрочить ориентацию на сервисы, можно установить дополнительные принципы.

- Раскрытие бизнес-функций с помощью модели ресурсов REST (см. врезку «Ресурсоориентированная архитектура» ниже) вместо сложных систем. Это требует большой работы и направлено на понимание бизнес-концепций и их свойств, правильное именование отношений, соответствующее проектирование схемы, верное определение идентичности и уникальности и т. д.
- Размещение данных как можно ближе, чтобы избежать оркестрации на уровне данных.
- Сохранение предметной логики в вашей области.
- Создание сервисов для потребителей с использованием правильного уровня детализации¹.
- Упрощение и использование современных общественных стандартов.
- Разрешение оркестрации сервисов (объединение нескольких конечных точек API в одну) только в пределах области (в ограниченном контексте).
- Единообразное использование идентификаторов областей, чтобы области могли распознавать взаимосвязи между ресурсами.

РЕСУРСООРИЕНТИРОВАННАЯ АРХИТЕКТУРА

В разработке программного обеспечения существует шаблон, называемый *ресурсоориентированной архитектурой* (*resource-oriented architecture, ROA*). ROA — это особый набор рекомендаций для архитектуры REST. Данные моделируются как коллекции однородных предметных областей или ресурсов, поэтому логически связанные данные объединяются в коллекции или ресурсы. Каждый ресурс — это объект, который идентифицируется с помощью URI. Например, ресурс «клиенты» может возвращать всю информацию о клиентах, включая имена, адреса и контактную информацию. Ресурсы данных в этом шаблоне играют роль существительных и должны обнаруживаться и объясняться без привлечения дополнительных ресурсов (SOA, напротив, обычно ориентирована на глаголы).

¹ Филипе Ксименес (Filipe Ximenes) и Флавио Хувенал (Flávio Juvenal) написали статью в своем блоге (<https://oreil.ly/sEEyW>) о том, как API на основе REST могут представлять ресурсы, которые правильно отформатированы, связаны и помечены версией.

Следование этим рекомендациям и набору принципов поможет вам сделать вашу архитектуру API масштабируемой, а также позволит областям оставаться изолированными и использовать при этом другие сервисы, не требуя от вас слишком больших усилий.

Новая роль ESB

Установив различия новых шаблонов декомпозиции, мы можем критически взглянуть на ESB. Теперь ESB потребуется только для тяжелой обработки, во всех остальных случаях лучше использовать шлюз API. Здесь мы можем сделать вывод, что ESB будет применяться исключительно для решения проблем модернизации устаревших систем и серьезной технической интеграции, как показано на рис. 4.7.

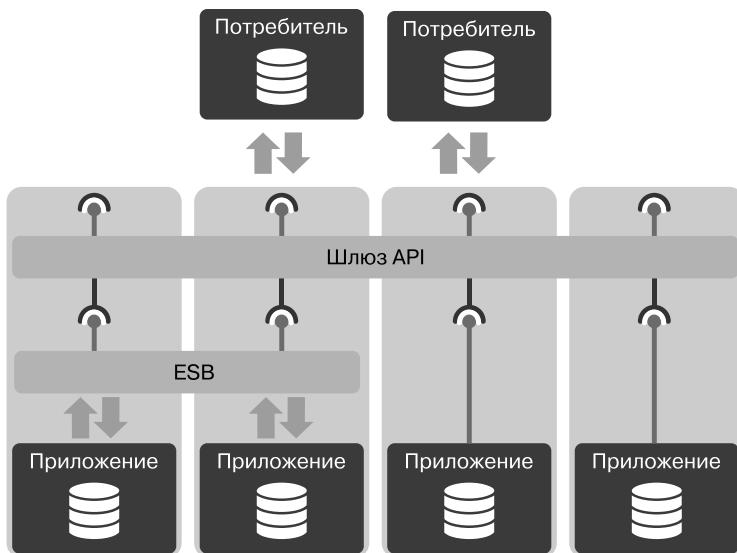


Рис. 4.7. Используя шлюз API для коммуникаций, мы предлагаем применять ESB для декомпозиции устаревших систем

Переведя ESB на один уровень ниже, ближе к устаревшим областям, и переложив ответственность на эти области, мы заставляем их требовательнее относиться к своим приложениям. Сегодня многие базы данных и даже устаревшие платформы легко можно инкапсулировать с помощью дополнительных технологий API. z/OS Connect (<https://oreil.ly/T67xE>) от IBM, например, можно использовать для сокрытия сложности устаревших мейнфреймов с помощью REST API. SQL Server (<https://oreil.ly/oKFMI>) от Microsoft и Oracle с REST Data

Services имеют аналогичные способы предоставления REST API для своих платформ, не требуя ESB.

Еще один способ модернизации традиционных или унаследованных приложений – развернуть небольшой набор функций или микросервисов рядом с БД или приложением. Эти функции работают аналогично инкапсуляции. Большим преимуществом этих шаблонов является то, что оптимизация данных и создание услуг являются заботой областей. Области вынуждены рассматривать свои API как собственные продукты.

Новое позиционирование шлюза ESB и API требует такой модели управления, чтобы области серьезно относились к своим обязанностям. *Контракты на обслуживание*, которые рассматриваются как контракты на поставку данных (см. подраздел «Контракты на поставку данных и соглашения о совместном их использовании» на с. 63), являются важной частью управления API.

Контракты на обслуживание

Из-за переноса ответственности ближе к областям создание и обслуживание сервисов выполняются не в центральной группе, а в самих областях. В случаях, когда команд много, необходимо, чтобы все они согласовали спецификации и проект API. Вот тут-то и появляются API или контракты на обслуживание¹.

API или контракт на обслуживание – это документ, который фиксирует организацию API (структуру, протоколы, версии, методы и т. д.) и может использоваться для заключения соглашений между поставщиками и потребителями услуг. Распространенной формой создания контрактов API является *спецификация OpenAPI* (<https://oreil.ly/cZ9De>) (ранее известная как *спецификация Swagger*). Преимущество документирования этих контрактов в коде состоит в том, что и люди, и системы могут интерпретировать спецификации. Поставщики услуг смогут получать процедуры тестирования от потребителей, поэтому любое изменение можно протестировать и проверить работоспособность сервиса. Pacto (<https://oreil.ly/V-1Ws>) и Pact (<https://oreil.ly/qdFTx>) – две платформы с открытым исходным кодом, которые могут помочь в создании и сопровождении этих контрактов.

Обнаружение сервисов

Чтобы отслеживать службы, я рекомендую вести список всех (бизнес-) API в реестре сервисов. *Реестр сервисов* – это инструмент, который хранит всю важную информацию об API в центральном репозитории, что способствует повторному использованию сервисов и предотвращает разрастание API (рис. 4.8).

¹ Мартин Фаулер подробно рассказывает о преимуществах контрактов, ориентированных на потребителя (<https://oreil.ly/Qcn-3>).

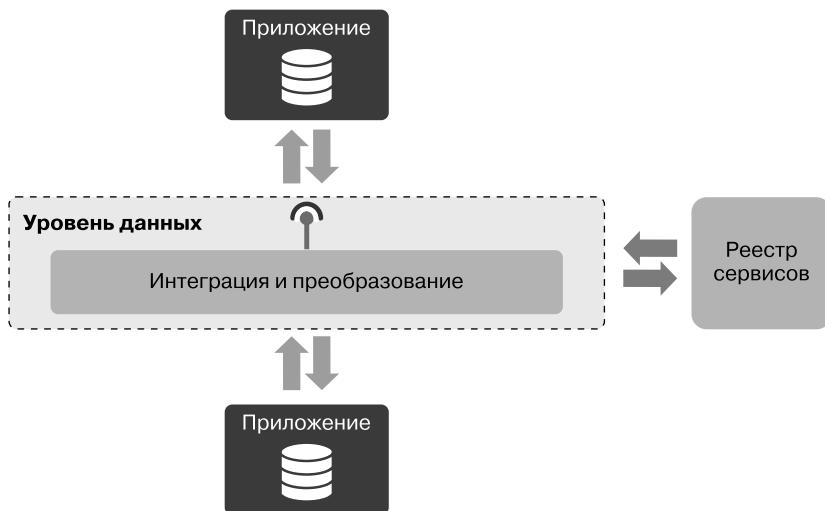


Рис. 4.8. Реестр сервисов хранит информацию обо всех сервисах: какие API доступны и кем используются. В интегрированной модели с несколькими шлюзами API особенно важны метаданные и регистрация API

Еще одно преимущество реестр обеспечивает возможность *обнаружения сервисов*¹. Это позволяет отслеживать местоположение API в сети, их состояние (доступность, время отклика), номера версий, количество потребителей и т. д. Eureka от Netflix (<https://oreil.ly/qaOOx>) является примером такого реестра. Реестр сервисов — отличный инструмент, особенно в распределенной среде с несколькими поставщиками облачных сервисов и множеством шлюзов API.

Микросервисы

Теперь, разрушив нашу сложную и тесно интегрированную сервис-ориентированную архитектуру и провозгласив курс на отделение сервисов от предметных областей, мы готовы к встрече с новой тенденцией — микросервисами. *Микросервисная архитектура* — это архитектурный шаблон, который разбивает приложения на еще более мелкие и управляемые части. Некоторым нравится использовать термин «*слабосвязанные сервисы*», который сразу указывает на пересечение с SOA.

¹ Обнаружение служб также используется в микросервисной архитектуре (<https://oreil.ly/8Bmt7>).

Для того чтобы лучше понять, что такое микросервисы, давайте быстро выясним, что такое приложение. Приложение — это компьютерная программа, предназначенная для выполнения группы скоординированных функций или задач. Приложения обычно имеют три уровня: представление, логику и данные.

Способы разработки приложений и использования базовой технической инфраструктуры в организациях изменились за последние пару лет. Более поздние подходы подразумевают разделение приложения на несколько частей. Это позволяет работать с отдельными частями приложения, что обеспечивает более высокую маневренность и возможность гибко масштабировать отдельные части. Этот подход проиллюстрирован на рис. 4.9.



Рис. 4.9. Сравнение традиционного трехуровневого приложения с микросервисами. В приложении с микросервисами ожидается, что каждый микросервис будет иметь свои данные. Синхронная связь между компонентами обычно осуществляется через API

Еще одно преимущество архитектуры микросервисов — в возможности разрабатывать, тестировать и развертывать части приложения, известные как микросервисы, индивидуально и независимо. В этой архитектуре каждый компонент выполняется в отдельном процессе, что делает ее гораздо более масштабируемой. Вместо масштабирования целого приложения теперь можно масштабировать отдельные компоненты. Стоит упомянуть и другие характеристики.

- Каждый микросервис может иметь свою среду выполнения, библиотеки, инструменты и функции. Они не должны быть одинаковыми. Можно смешивать Python, Java, JavaScript, PHP и вообще все, что угодно.
- *Контейнеризация* или использование контейнеров, которые несут облегченную среду выполнения, — популярный способ развертывания.

- Каждый микросервис обычно хранит свои данные в выделенной БД¹. Используемые БД могут быть разными для разных микросервисов.
- Услуги логически организованы и управляются в ограниченном контексте.
- Связь с другими микросервисами обычно осуществляется с помощью упрощенных механизмов, таких как JSON с REST, gRPC и Thrift.

Последний пункт вызывает наибольшую путаницу. Внутри микросервисов есть некоторые общие компоненты, которые нужны для поддержки связи. Одним из них является API-шлюз.

Роль API-шлюза в микросервисах

Микросервисы обычно взаимодействуют так же, как приложения внутри SOA. Еще больше сбивает с толку то, что при разработке больших или сложных приложений на основе микросервисов для разделения отдельных микросервисов используются шлюзы API. Кроме того, несколько микросервисов из одной предметной области можно изолировать от других областей с помощью API-шлюза. Шлюз API в архитектуре этого типа предлагает те же преимущества, что и в SOA, как показано на рис. 4.10.

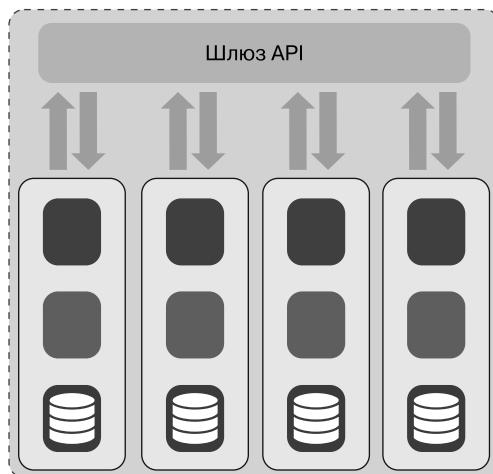


Рис. 4.10. Шлюз API используется для взаимодействий между микросервисами

¹ Хочу добавить, что (это все еще остается спорным, но, вероятно, об этом стоит упомянуть) при наличии одной базы данных на микросервис допускается определенная степень избыточности.

Тенденция к использованию микросервисов сочетается с другой важной тенденцией: масштабируемыми контейнерными и бессерверными платформами. Kubernetes (<https://kubernetes.io/>) — популярная контейнерная платформа с открытым исходным кодом — отлично справляется с крупномасштабными рабочими нагрузками микросервисов. Каждый микросервис обычно развертывается внутри как отдельный сервис и работает как контейнер. Запуск сотен или тысяч микросервисов на такой платформе не является чем-то необычным.

Функции

Функция как услуга (function as a service, FaaS) — еще одна модель организации микросервисных архитектур. Это относительно новая категория сервисов облачных вычислений, которая помогает выполнять отдельные «функции, действия или части бизнес-логики». FaaS позволяет инженерам разрабатывать, запускать прикладные функции и управлять ими без сложностей, связанных с созданием и обслуживанием инфраструктуры, обычно присущих этому процессу. В такой вычислительной модели функции развертываются по запросу и с оплатой по факту использования. Эта модель сокрытия инфраструктуры также известна как *бессерверная*¹.

У каждой функции обычно есть API, соответствующий этой функции. API в этой модели нужны для вызова или активизации функций. Принципы работы этих функций и соответствующих API одинаковы для всех других API. При использовании в границах предметной области функции могут вызываться напрямую. При пересечении границ бессерверные API должны подчиняться репозиторию API, заключая контракты и регистрируясь в нем.

Хотя функции маскируются шлюзом API, прослеживается значительное совпадение с событийно-ориентированными архитектурами (см. главу 5). Функции во многих случаях используются в асинхронных сценариях. Основная причина в том, что иногда функции работают медленно. В облачных вычислениях большая часть быстрых ресурсов зарезервирована для дорогих или ресурсоемких приложений. Оставшиеся «дешевые» ресурсы отдаются бессерверным функциям. Из-за этого у функций не очень предсказуемое время отклика. Другая причина сходства функций с событийно-ориентированной архитектурой заключается в том, что функции обычно выполняются асинхронно. Они могут использовать очереди сообщений, чтобы сохранить свои результаты и дождаться результатов других процессов (функций).

¹ Бессерверное состояние не означает, что серверов не существует. Это модель выполнения облачных вычислений. В бессерверном режиме все детали инфраструктуры скрыты от пользователя. Мартин Фаулер хорошо это описывает (<https://oreil.ly/y2PnK>).

МИКРОСЕРВИСЫ И БЕССЕРВЕРНЫЕ АРХИТЕКТУРНЫЕ ШАБЛОНЫ

Микросервисы и бессерверные вычисления можно сочетать для создания надежных и масштабируемых приложений. Здесь рассматриваются некоторые хорошо известные шаблоны.

- Web-Queue-Worker («Веб-интерфейс — очередь — рабочая роль») (<https://oreil.ly/II3Gi>) — это стиль, согласно которому приложение имеет веб-интерфейс, обрабатывающий HTTP-запросы, и (на основе API) рабочие процессы, выполняющие тяжелые вычисления или продолжительные операции.
- Strangler («Душитель») (<https://oreil.ly/uC1XR>) — шаблон для постепенного избавления от устаревших систем или вывода их из эксплуатации. Вместо создания нового приложения с нуля унаследованное приложение постепенно преобразуется в небольшие микросервисы. По мере готовности новых функций старые компоненты выводятся из эксплуатации, а интерфейс корректируется без уведомления пользователей.
- Fan-Out/Fan-In («Ветвление/слияние») (<https://oreil.ly/QrZUs>) — шаблон одновременного выполнения нескольких функций и последующего агрегирования результатов.
- Шаблон Gateway Aggregation («Агрегирование в шлюзе») (<https://oreil.ly/ttm6J>) используется для объединения нескольких отдельных запросов в один с целью избежать издержек на связь, когда множеству мелких служб требуется объединить большой объем данных.
- Шаблон Gateway Routing («Маршрутизация в шлюзе») (<https://oreil.ly/DdJyf>) применяется для маршрутизации запросов к нескольким службам с использованием одной конечной точки. Этот шаблон применяется для передачи запросов нескольким службам через единственную конечную точку.

Джереми Дейли (Jeremy Daly) ведет обзор (<https://oreil.ly/JPtRw>) с помощью примеров проектирования и шаблонов микросервисов.

Сервисная сетка

Для управления микросервисами на крупномасштабной платформе существует еще один шаблон, который называется сервисной сеткой. *Сервисная сетка* — это уровень (прокси), предназначенный для поддержки взаимодействий между сервисами. Сервисная сетка и шлюз API во многом пересекаются, хотя и различия между ними тоже есть. Оба обеспечивают декомпозицию, мониторинг, обнаружение, маршрутизацию, аутентификацию и регулирование. Основное различие заключается во внутренних взаимодействиях между сервисами. Маршрутизация в сервисной сетке осуществляется не извне, а изнутри, в пределах внутренних границ сервиса, в которых работают микросервисы. На рис. 4.11 показаны обе модели взаимодействий.

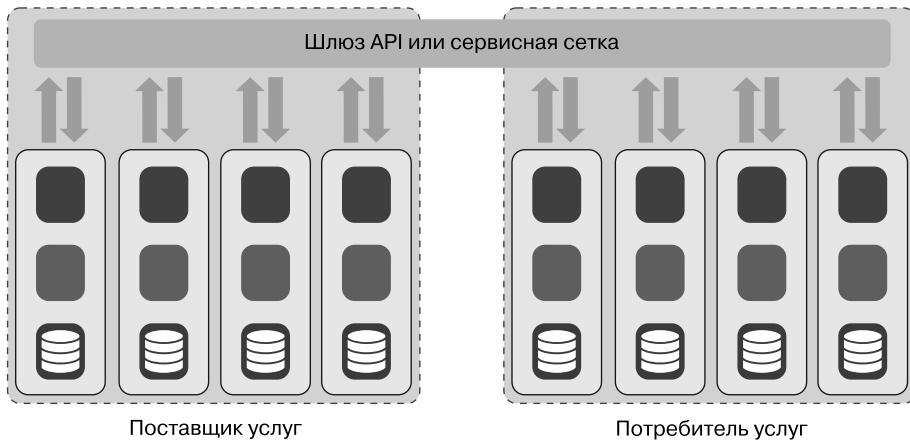


Рис. 4.11. При увеличении количества микросервисов становится важным разделение разных областей

Слева изображены все микросервисы — поставщики услуг. Все внутренние взаимодействия между поставщиками обрабатываются сервисной сеткой (действующей как внутренний шлюз API). Когда нужна связь с другой стороной, например микросервисами или приложениями из другой области или окружения, трафик перенаправляется вовне. Для этого можно использовать сервисную сетку или дополнительный шлюз API. В последнем случае шлюз API выполняет внешнюю маршрутизацию, аутентификацию и принимает входящий трафик извне, а сервисная сетка заботится об управлении «внутренней» архитектурой приложения.



Существует несколько видов шлюзов API, решающих задачи управления распределенными данными в вашей архитектуре. Некоторые реализации поддерживают распределенную архитектуру, основанную на группах шлюзов API в административной среде. Например, Apigee (<https://oreil.ly/DEF55>), Kong (<https://oreil.ly/Xi7mW>), MuleSoft (https://oreil.ly/C_cNk) и WSO2 (<https://wso2.com/>) позволяют развернуть несколько узлов, чтобы дать возможность управлять всеми внутренними и внешними API с центральной панели. Преимущество в том, что все эти экземпляры управляются как одно целое с использованием одинаковой конфигурации и политик.

Вы можете задаться вопросом: обязательно ли использовать и то и другое? Я думаю, что лучше всего использовать их вместе. Но по мере развития сервисных сеток и шлюзов API я ожидаю, что их возможности будут расширяться и эти две концепции объединятся. Однако есть одно архитектурное различие, когда сервисная сетка используется для разделения предметных областей предприятия.

Если основная роль сервисной сетки заключается в оркестрации микросервисов в определенной области, то логично организовать оркестрацию областей через шлюз API. Прагматическое решение — развернуть дополнительный шлюз API — в виде микросервиса. Apigee, например, использует микрошлюз (<https://oreil.ly/8PO9Q>), который можно развернуть непосредственно в Kubernetes.

Границы микросервисов

Глядя на рис. 4.11, можно заметить логические границы вокруг микросервисов поставщиков и потребителей услуг. Если на одной платформе работает много микросервисов, важно провести логические линии. Разграничение и декомпозиция сервисов играют важную роль, помогая областям управлять своей сложностью и зависимостями от других областей. Для определения границ сервисов используется модель предметно-ориентированного проектирования. С изменением ограниченного контекста будут меняться и его границы, а сервисы должны отделяться друг от друга. Следуя этой логике, мы можем отделить микросервис дважды.

Первое отделение происходит в границах области на уровне внутренней связи между сервисами: один микросервис отделяется от другого в той же области. Второе отделение происходит на уровне взаимодействий между областями: микросервисы, взаимодействующие с микросервисами из другой предметной области, снова отделяются. В первом случае проектирование API носит более технический характер. Эти API не предназначены для прямого доступа к другим областям и не публикуются в каталоге сервисов. В ситуации взаимодействий между областями API должны оптимизироваться для потребителей, поэтому необходимо соблюдать принципы проектирования, оптимизированного для чтения.

Микросервисы в эталонной архитектуре API

Чтобы вписать архитектуру микросервисов в общую архитектуру API, давайте посмотрим на общую картину.

Во-первых, мы можем заметить, что шлюз API играет важную роль в архитектуре для декомпозиции всех взаимодействий между областями. Во-вторых, сервисы, осуществляющие взаимодействия между областями, должны зарегистрироваться в репозитории. Наконец, ко всем этим API применяются принципы оптимизации для чтения.

На рис. 4.12 показаны три различных шаблона проектирования.

1. Вверху находятся современные приложения, которые могут быть доступны напрямую через шлюз API. В этом шаблоне приложение использует современные средства взаимодействий, такие как REST/JSON.

2. Посередине находятся унаследованные приложения, требующие дополнительной развязки через ESB. Сервис ESB может быть заключен в шлюз API.
3. Внизу находятся микросервисы, обменивающиеся данными через сервисную сетку API, которые должны быть доступны извне, отделяются с помощью шлюза API.

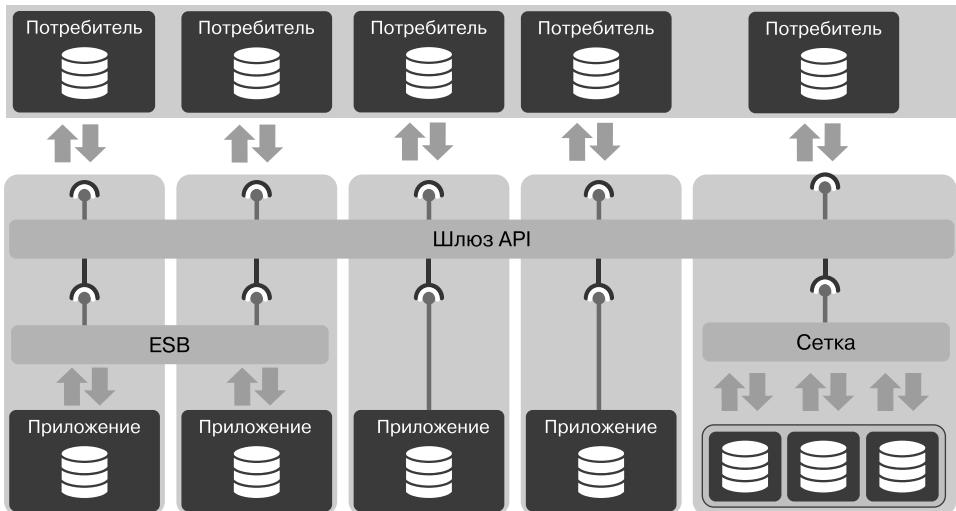


Рис. 4.12. Архитектура API не исключает микросервисы, и наоборот. Сервисную сетку или шлюз API, используемые в микросервисах, можно идеально сочетать с архитектурой API

Группировка приложения и сервисов на уровне области соответствует тому, что было описано в главе 2. Помещая ESB, шлюз API и сервисную сетку под крыло архитектуры API, мы создаем единую стеклянную панель, благодаря которой можно наблюдать и контролировать все взаимодействия между сервисами. Каждая область изолирована, использует свой предметный язык для взаимодействий внутри и единый язык коммуникации с другими областями. Единственное исключение из этого правила — API, которые будут открыты для «внешних экосистем».

Коммуникация между экосистемами

В главе 1 я упомянул, что многие организации изучают новые бизнес-модели на основе API и интернета. Разные компании пришли к выводу, что им нужно тесно сотрудничать с другими цифровыми компаниями, чтобы расширяться, внедрять инновации, менять курс или просто оставаться конкурентоспособными. Ожидается, что эта тенденция сохранится и заставит вашу архитектуру четко

разделить *внутреннюю* (сервис) и *внешнюю* (сервис) связь с потребителями¹. На это есть три важные причины.

- Язык организации, используемый внутри компании, находится под ее контролем, но он не всегда совпадает с языком, используемым другими компаниями. Формат и структура API часто диктуется другими сторонами или нормативными актами. PSD2, как я упоминал в главе 1, продиктован европейским банковским сектором и вынуждает банки строго организовывать свои API по счетам, партнерам, транзакциям и т. д. Как следствие, при повторном использовании существующих внутренних сервисов часто требуется повторный перевод.
- API, доступные в публичной сети, требуют дополнительного внимания. Внешние стороны не всегда заслуживают доверия или известны. Чтобы защитить API, необходим мониторинг и управление. Еще не помешает иметь поставщиков услуг идентификации для безопасного доступа.
- Возможно, вам захочется отрегулировать передачу данных (ограничить количество вызовов) на основании заключенных коммерческих соглашений. Внешние API, предоставляющие услуги на коммерческой основе, часто имеют план потребления и оплачиваются в зависимости от количества вызовов. Мониторинг API может быть разным, потому что нужно различать разных пользователей API.

Для взаимодействия внешних и публичных веб-сервисов требуется дополнительный уровень в архитектуре, который находится между внутренними сервисами и внешними потребителями. Этот уровень API показан на рис. 4.13.

Могут ли внутренний и внешний шлюзы API быть одинаковыми? Технически можно использовать одну платформу как для внутренней, так и для внешней маршрутизации трафика. Однако ключевую роль здесь играет бизнес-стратегия API. Успех открытых API зависит от их способности привлекать внешних разработчиков. Целью открытых API является разработка ценных приложений, которые разработчики и пользователи действительно захотят применять. Другая платформа позволит быстрее внедрять инновации, поскольку она будет слабее связана с внутренней средой.

Еще одна причина отделения внутренней корпоративной коммуникации от внешнего трафика — это различные меры безопасности. Внешнему трафику нужна дополнительная регистрация, защита от DDoS-атак (<https://oreil.ly/OjLEU>), услуги различных поставщиков идентификации и т. д.

Можно предположить, что внешние API и взаимодействие с экосистемой сильно повлияют на вашу стратегию API, ведь для этого потребуется разли-

¹ Здесь «внутренний» не относится к внутренним API приложения. В этом случае имеются в виду внутренние средства в границах организации или предприятия.

чать внутренние и внешние взаимодействия. Еще одна цель, которая влияет на архитектуру, — это коммуникация по каналам для предоставления клиентам широкого спектра услуг.

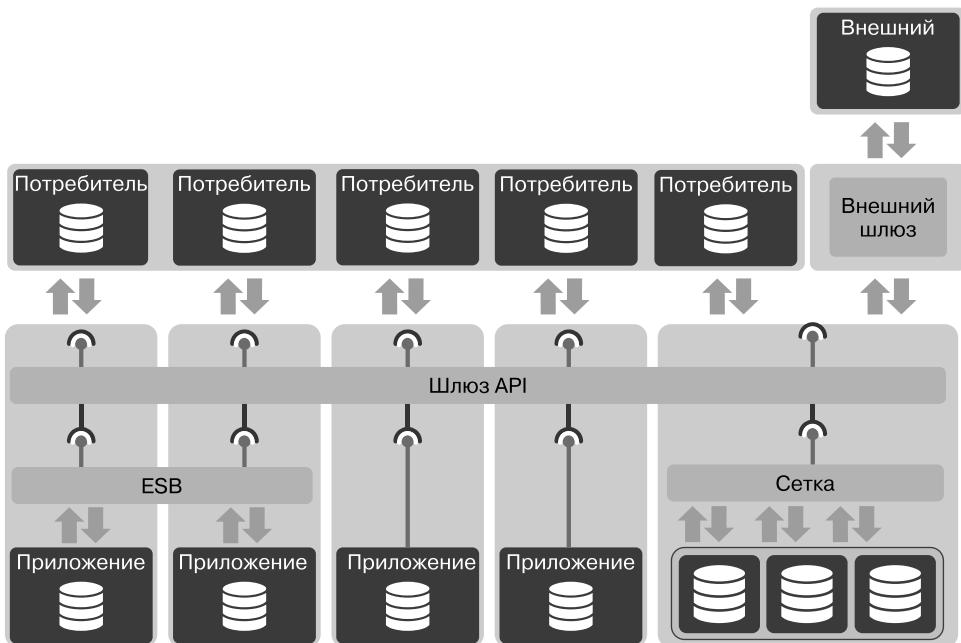


Рис. 4.13. Внешний шлюз API выполняет особую роль, осуществляя маршрутизацию и защищая API для внешних потребителей

Каналы связи на основе API

Одна из ключевых задач — использовать API для веб-каналов, где общение происходит с клиентами, а контент предоставляется другим сторонам. API являются основным строительным блоком архитектуры API. API важны, поэтому нам нужно различать «межобластное» и «канальное» взаимодействие API.

Связь между областями — это общий API, о котором мы говорили до сих пор: связь между приложениями в разных областях. Пример — вызов API системы финансовых платежей из системы CRM.

Связь по каналу отличается, потому что здесь API используются для прямой связи с конечными пользователями. Это в большей степени связь между людьми, чем между системами. Взаимодействия такого вида могут протекать через

интернет, смартфон и компьютер, чат-боты, игры, видео и системы распознавания речи. Зная, что онлайн-каналы часто являются товарным знаком компании, важно предложить отличный пользовательский интерфейс и удобство работы, что требует дополнительных компонентов в архитектуре для проверки прав, обработки и проверки запросов, управления состоянием, кэширования, защиты и т. д. Эти компоненты, показанные на рис. 4.14, являются частью предметной области канала и тесно взаимодействуют со шлюзом API.

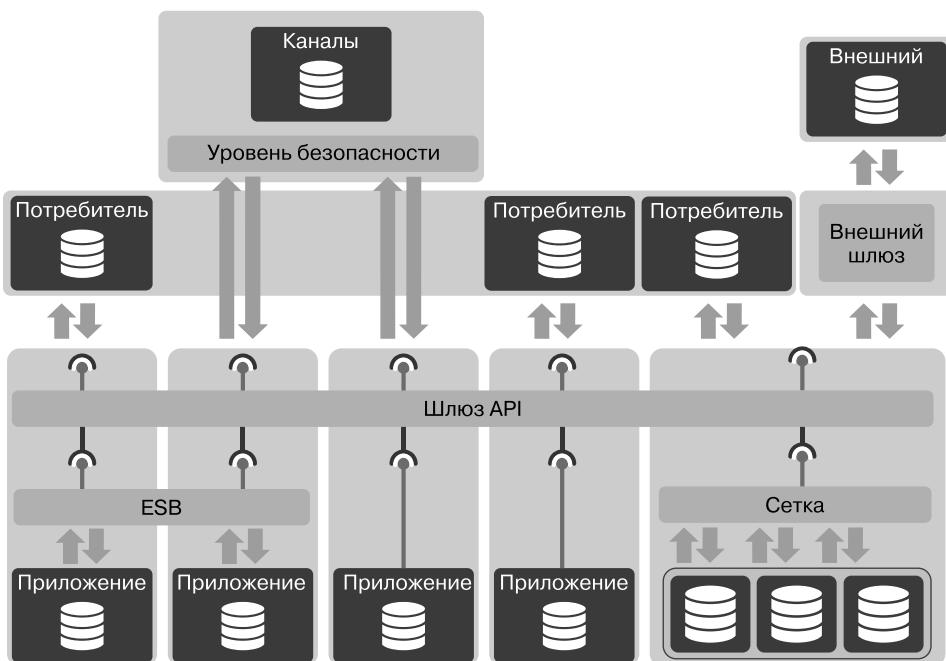


Рис. 4.14. Канальный уровень имеет дополнительный уровень безопасности для веб-коммуникаций и может содержать собственные компоненты

Причина, по которой эти компоненты находятся в области канала, ближе к конечному пользователю, в том, что улучшение пользовательского опыта само по себе является отдельной задачей. Безопасность веб-приложений отличается от безопасности API. Пользовательский интерфейс должен быть быстрым. Это требование может сильно повлиять на способ взаимодействия с API. Выполнение большого количества вызовов API к ресурсам на стороне сервера может замедлить взаимодействие с пользователем. Простой список выбора стран лучше хранить в выделенном (кэшированном) компоненте, рядом с конечными пользователями, чем каждый раз извлекать его по сети через

шлюз API. Нужно совершать меньше вызовов и более эффективно использовать получаемые данные. Поэтому необходимо добавить в свою архитектуру дополнительные функции.

GraphQL

Типичным компонентом окружения канала является сервис GraphQL (<https://oreil.ly/JB4OA>). GraphQL — это язык запросов для выбора именно тех полей, которые нужно получить из API. Он снижает накладные расходы на сетевые взаимодействия и может создавать комбинации ресурсов при запросе и объединении разных конечных точек API.



GraphQL допускает *объединение схем* (<https://oreil.ly/zeyJy>) — их организацию в единый граф данных, чтобы обеспечить единый интерфейс для запросов ко всем вашим наборам данных. Но не попадайтесь в ловушку создания корпоративной модели данных или модели центрального шлюза. Сборка схемы и интегрированная конструкция по своей сути зависят от особенностей использования области конкретным клиентом. Желающим узнать об этом больше, я рекомендую прочитать *Visual Design of GraphQL Data: A Practical Introduction with Legacy Data and Neo4j* Томаса Фризендала (Thomas Frisendal).

Сервисы GraphQL в архитектуре API могут быть развернуты только в области канала или как средство связи между областями. В последнем случае необходимо соблюдать все принципы, описанные в главе 2: конечные точки GraphQL должны следовать одному и тому же управлению API, включая обязательные контракты на обслуживание и правила обратной совместимости.

Внутренние интерфейсы для внешних

Еще один способ улучшить взаимодействие с пользователем и удовлетворить потребности клиентов внешнего интерфейса — это управление пользовательским интерфейсом и логикой обработки с помощью дополнительного уровня или компонентов. Этот уровень можно оптимизировать для нужд определенных внешних интерфейсов. Например, мобильные или одностраничные приложения могут использовать другие оптимизации и другие компоненты, нежели настольные приложения. В этом уровне тоже можно применять GraphQL.

Сэм Ньюман (Sam Newman) описывает философию проектирования, заключающуюся в использовании сервисов и применении оптимизаций и абстракций без общего представления или привязки друг к другу, как *backends for frontends* (внутренние интерфейсы для внешних) (<https://oreil.ly/cxfW6>).

Метаданные

Управление метаданными API — важный аспект архитектуры, помогающий потребителям находить сервисы и повторно использовать их. Начать можно с публикации каждого API, включая их атрибуты и описания, в центральном репозитории. У многих современных платформ управления API есть такой репозиторий. Но если вы не хотите иметь дело с коммерческими поставщиками, можно использовать решение для управления API с открытым исходным кодом. Umbrella API (<https://oreil.ly/jlj6r>), Swagger (<https://oreil.ly/EG4BI>) и Kong (<https://oreil.ly/XM349>) — вот некоторые популярные примеры.

Особенно важно отслеживать все API в распределенной среде, где развернуто несколько шлюзов API, ESB и сервисных сеток. Однако это даст положительный результат, только если каждая функция интеграции будет подключена к репозиторию метаданных (рис. 4.15). Регистрация позволит узнать, какие сервисы каким областям принадлежат и какие области взаимодействуют друг с другом. Такое понимание станет вашим преимуществом в управлении общим ландшафтом.

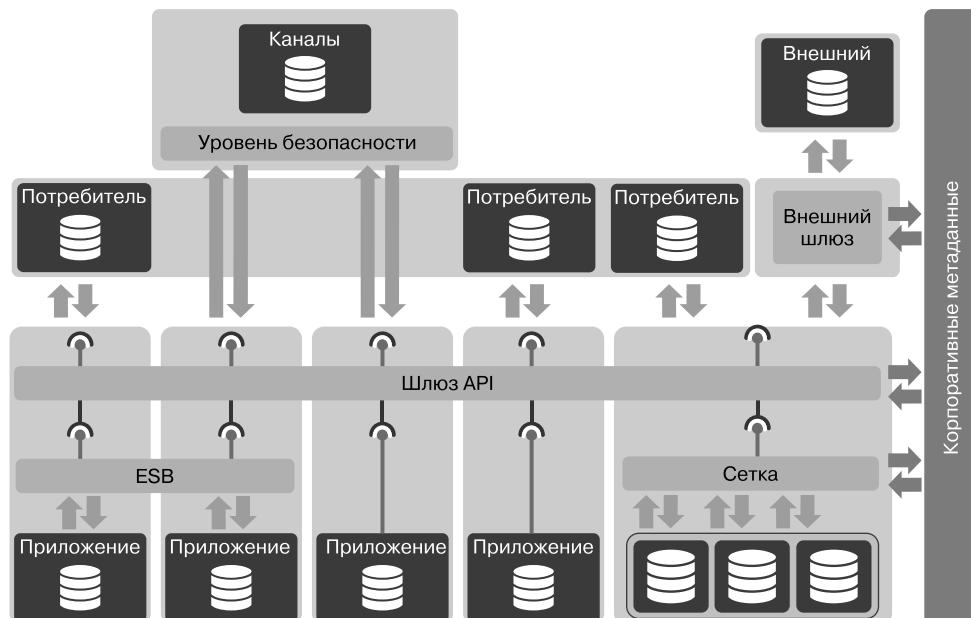


Рис. 4.15. Уровень метаданных дает представление обо всех взаимодействиях API

Еще одна область, в которой помогают метаданные, — это контракты на обслуживание между поставщиками и потребителями услуг. При использовании

Pacto (<https://oreil.ly/bG7Ch>) и Pact (<https://oreil.ly/mCGsE>) вы можете рассмотреть возможность хранения всех контрактов, включая интеграционные тесты, в центральном репозитории. Прозрачность контрактов позволяет понять их сложность и взаимосвязь.

МИКРОСЕРВИСЫ И МЕТАДАННЫЕ

Должны ли микросервисы предоставлять метаданные? Безусловно! Тысячи микросервисов могут работать на крупномасштабной платформе. Важно знать, кому какой сервис принадлежит, чем он занимается и какие данные производит. Чтобы позволить микросервисам заявить о себе, рассмотрите возможность добавления меток метаданных в среду выполнения или вызова API после развертывания микросервиса. Чтобы упростить работу групп, вы можете сделать вызов API частью базовых образов контейнера. Еще один способ помочь командам — предоставить фрагменты кода или основные шаблоны.

Последней сферой, где важны метаданные, является сфера безопасности и идентификации. Мы рассмотрим ее в главе 7, а пока просто имейте в виду, что метаданные определяют, какие пользователи и к каким сервисам имеют доступ.

Использование RDS для чтения в реальном времени и активного чтения

Архитектура API не является самостоятельной, ее можно комбинировать с архитектурой RDS. Как только RDS достигнут достаточной производительности, их можно использовать для обработки всех запросов чтения, подготовив RDS или ориентированные на чтение микросервисы, которые действуют как кэши и хранят данные локально, способствуя потреблению данных.

Такой подход позволяет реализовать шаблон CQRS (см. подраздел «Что такое CQRS» на с. 77), как показано на рис. 4.16, для организации взаимодействий в реальном времени. Шлюз API как часть архитектуры API будет отделять и направлять запросы на обслуживание. В этой модели запросы на чтение строго согласованных данных, для которых гарантируется возврат самых актуальных результатов, а также команды, по-прежнему направляются в операционные системы. Остальные запросы на чтение данных, которые будут *согласованы по прошествии некоторого времени*, передаются в RDS¹.

¹ Вернер Фогельс (Werner Vogels), технический директор Amazon, прекрасно объясняет, в чем заключается согласованность по прошествии некоторого времени (<https://oreilly/kWOZz>).

Вместо развертывания API в RDS можно развернуть ориентированные на чтение микросервисы, действующие как кэши и напрямую поставляющие данные потребителям. Каждый микросервис в этой архитектуре хранит часть ценных данных в собственном хранилище и обслуживает его через API. Можно провести параллель с ресурсоориентированной архитектурой, в которой каждый микросервис кэширует данные для одного конкретного ресурса. Опять же в масштабируемой архитектуре API различают все команды и запросы.

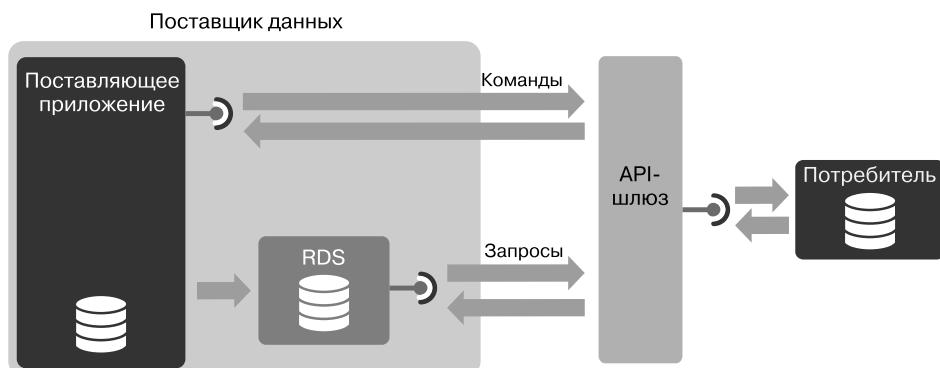


Рис. 4.16. CQRS для чтения в реальном времени: все запросы на получение строго согласованных данных и команды обслуживаются архитектурой API, а остальные — архитектурой RDS

Как вы узнали из главы 3, с помощью CQRS можно значительно улучшить масштабируемость и производительность архитектуры и оптимизировать ее для более эффективной обработки запросов. Этот подход работает и в распределенной среде, где операционная система находится на одной стороне сети, а система, посылающая запросы, — на другой. Когда нагрузка очень высока, CQRS — единственный способ решить проблемы управления распределенными данными.

Итоги главы

В быстро меняющемся корпоративном ландшафте можно рассматривать архитектуру API, описанную в этой главе, как способ обновить стратегию API. Основы сервис-ориентированной архитектуры остаются неизменными, но получают более четкие обязанности, современные шаблоны, принципы и более гибкую архитектуру.

Благодаря своей REST-основе архитектура API — отличный способ распространения всех данных. Вы можете развернуть несколько шлюзов API и сер-

висных сеток одновременно, сохраняя при этом полный контроль с помощью метаданных. Потенциально каждая команда или область могут использовать собственный шлюз API для предоставления услуг. Вы можете контролировать ситуацию, используя репозитории метаданных, в которых хранится информация о том, какие сервисы доступны и как они применяются.

Модульность и модель запрос/ответ архитектуры API также позволяют создавать высокомодульные микрорешения для конкретных случаев использования. Это делает архитектуру хорошо масштабируемой. Но имейте в виду, что вы должны следовать тем же принципам архитектуры RDS, которые обсуждались в главе 2. API следует рассматривать как продукты, используемые для генерации, отправки и получения данных в реальном времени. Они должны находиться в вашем владении и оптимизироваться для удобочитаемости и простоты применения, а также использовать единый язык предметной области, но без включения логики других областей.

В следующей главе мы рассмотрим потоковую архитектуру, в которой основное внимание уделяется асинхронному обмену сообщениями и коммуникации, управляемой событиями.

ГЛАВА 5

Управление событиями и ответами: потоковая архитектура

В этой главе мы обсудим потоковую архитектуру, управляемую событиями. Она является наиболее сложной, потому что пересекается с архитектурами RDS и API. Мы рассмотрим такие вещи, как асинхронные взаимодействия, событийно-ориентированные архитектуры, и такие технологии, как Kafka, а также модели согласованности, типы событий и многое другое. К концу этой главы вы получите полное представление о том, что событийно-ориентированная архитектура может привнести в вашу организацию.

Знакомство с потоковой архитектурой

Архитектура потоковой передачи заимствует свое название от *потоковых данных* (<https://oreil.ly/2T5XG>) или *обработки потоков событий* (<https://oreil.ly/sz8lb>). Оно означает непрерывную обработку и анализ данных из разных источников в реальном времени. Такой шаблон имеет еще много названий (тонкие различия между ними будут объяснены в пункте «Сервисы потоковой аналитики» на с. 177): обработка событий, сложная обработка событий (complex event processing, CEP), аналитика в реальном времени, потоковая аналитика и потоковая аналитика в реальном времени.

Потоковая передача данных в реальном времени позволяет реагировать быстрее и лучше удовлетворять запросы клиентов. Более быстрые результаты делают выводы более актуальными. Обнаружив мошенничество в момент совершения, можно отреагировать немедленно. Вы можете видеть, откуда приходят посетители, взаимодействовать с ними и улучшать качество обслуживания их как клиентов, пока они остаются на сайте. Данные, доставляемые в потоке событий, могут иметь огромную ценность для организаций, а их источники могут сильно различаться и включать, например, устройства интернета вещей (Internet of Things, IoT), смартфоны, онлайн-игры, социальные сети и традиционные операционные системы.

В отличие от пакетной обработки и API, потоковая передача данных ориентирована на взаимодействие с событиями и реакцию на них. Мы говорим об асинхронном соединении приложений и систем. Потоковая передача имеет высокую пропускную способность и может использоваться как без отслеживания состояния, так и с ним. Она позволяет обмениваться информацией об изменениях, которые происходят в определенные моменты времени (то, что мы называем *событиями*), и перемещать состояние приложения между местоположениями. Это обеспечивает возможность создания распределенных, актуальных, материализованных представлений.

Для потоковой передачи данных используются платформы, позволяющие управлять потоком событий, комбинировать, фильтровать, агрегировать и сохранять события, а также запускать функции. Еще потоковая передача позволяет выявлять закономерности в поведении, быстро анализировать их и принимать решения. Когда такая платформа сохраняет данные, она приобретает черты сходства с хранилищем данных только для чтения (RDS), а кроме того действует подобно API. Естественно, шаблоны потоковой передачи требуют надежного управления данными. Например, одно событие может вызвать отправку множества сообщений или запросов большому количеству разных сервисов. С точки зрения управления данными мы также должны следить за тем, какой ответ принадлежит какому исходному сообщению.

Асинхронная модель событий имеет значение

Основное различие между архитектурой API и потоковой архитектурой — перевернутая диалоговая природа потоковой архитектуры. Вы слушаете, а не говорите. В этом и заключается ее сложность, потому что множеству пользователей чужд такой образ мышления. Когда что-то претворяется в жизнь, многим из нас нужна мгновенная реакция. Щелкая на ссылке, мы ожидаем, что новая страница появится немедленно. Взяв трубку и начав разговаривать, мы ожидаем, что человек на другом конце линии нас услышит. Синхронное общение, когда посыпается запрос и возвращается ответ, кажется нам естественным.

С другой стороны, реактивные взаимодействия требуют иного мышления. Хороший пример асинхронной связи — электронное письмо, в котором вы нажимаете кнопку «Отправить», а затем продолжаете работу, не дожидаясь ответа от собеседника. Приложения и платформы интеграции могут делать то же самое с событиями. Они отправляют событие или сообщение в очередь сообщений и ждут, пока другая система их заберет. Эта слабая связанность устраняет зависимость от ожидания немедленной реакции другой стороны. Так асинхронная связь делает масштабируемую архитектуру более устойчивой.



Клиенты потоковой передачи обычно открывают соединение с платформой и ждут, пока данные будут отправлены и доставлены. Соединение в этой модели имеет ограниченный срок жизни, в течение которого можно обмениваться данными. По истечении установленного времени все соединения должны быть закрыты, а ресурсы очищены. Более эффективная форма общения, чем многократное открытие и закрытие, — поддерживать соединение открытым как можно дольше. Этот метод называется длинным опросом (<https://oreil.ly/7Bwfw>).

Асинхронная передача данных не нова. Первые асинхронные системы обмена сообщениями и управления очередями относятся к 1960-м годам (https://oreil.ly/4r6Y_). Они также стали важной частью архитектур промежуточного программного обеспечения, ориентированного на сообщения (МОМ), таких как корпоративные сервисные шины (ESB), для облегчения асинхронной связи внутри SOA.

Очереди сообщений по-прежнему играют важную роль во многих компаниях. Есть большая разница между очередями сообщений, используемыми в архитектурах МОМ. Современные платформы обмена сообщениями масштабируют асинхронную модель за счет их распределенных (большие данные) и отказоустойчивых возможностей. Распределение вычислений и данных по множеству узлов означает, что, если одни узлы откажут, их задачи возьмут на себя другие. Здесь мы видим еще одно преимущество: масштабируемость для параллельной обработки. Каждый раз, когда вы удваиваете количество машин в распределенной архитектуре, вы удваиваете и объем вычислений/обработки. Фреймворки распределенной обработки, такие как Hadoop, работают так же.

Долгосрочное постоянное хранилище — это то, чем платформы потоковой передачи отличаются от традиционных систем очередей. Их распределенная природа и неизменяемые файловые системы, допускающие только добавление, позволяют сохранять огромное количество сообщений в течение длительного времени. Этим они существенно отличаются от традиционных очередей сообщений, которые уничтожают сообщения после доставки их потребителю. Современные платформы потоковой передачи событий также ведут себя как БД.

Как выглядят событийно-ориентированные архитектуры

Тенденция современных платформ потоковой передачи или обмена сообщениями открыла сферу так называемых *событийно-ориентированных архитектур* (event-driven architecture, EDA). Событийно-ориентированные архитектуры уделяют больше внимания событиям, чем сервисам. Событие можно описать как «изменение состояния» или «что-то, что произошло»: новый клиент регистрирует или размещает заказ, пилот приземляет самолет и т. д.



EDA не требует использования промежуточного ПО или брокера событий как посредника между производителями и потребителями событий. EDA также может быть реализована с использованием двухточечных интерфейсов, хотя это не рекомендуется, потому что при таком подходе архитектурой очень сложно управлять.

Каждый раз, когда происходит такое событие, генерируется сообщение (или *уведомление о событии*) и доставляется платформе потоковой передачи или обмена сообщениями. Ожидается, что каждое сообщение будет содержать данные о событии. Этот шаблон сопровождается эталонной моделью с немного другой философией проектирования. Используется либо топология посредника, либо топология брокера.

Топология посредника

Модель топологии посредника используется для проектирования архитектур, которым нужен некоторый уровень централизованного управления или координации для управления обработкой событий. Обычно она работает с посредниками и очередями сообщений, которые получают входящие сообщения и гарантируют, что каждое сообщение для данной темы или канала доставляется и обрабатывается потребителем ровно один раз. Эта топология обычно используется, когда требуется многоступенчатая обработка событий. В такой модели может быть много посредников и очередей сообщений. Каждая очередь обычно связана с определенной задачей.

Очереди сообщений (рис. 5.1) могут работать быстро, но, чтобы гарантировать обработку сообщений только один раз, они удаляются или помечаются флагком после того, как потребитель подтвердит получение. Очереди сообщений обычно используются, когда важно, чтобы каждое сообщение обрабатывалось только единожды или в нужной нам последовательности, например в транзакционных системах. Системы на основе очередей считаются двухточечными решениями, потому что повторное использование невозможно.

Топология посредника также пересекается с шаблонами сервис-ориентированной архитектуры, как обсуждалось в главе 4. Например, продолжительными процессами, требующими утверждения вручную, можно управлять с помощью дополнительного ПО для управления бизнес-процессами (*business process management*, BPM). BPM в этой модели действует как посредник и знает, какие процессы нужно приостановить, в какой очереди искать, где запрашивать утверждение вручную и какие еще службы следует вызывать.

Программная реализация топологии посредника в архитектуре — это *модель публикации-подписки*. В этой модели отправители сообщений, называемые *издателями*, не отправляют сообщения конкретным *получателям*, называемым

подписчиками, а распределяют сообщения по очередям, не зная, какие подписчики там могут быть. Реализовать эту топологию можно с помощью разного программного обеспечения. Известно, что корпоративные сервисные шины (ESB) хорошо справляются с посредничеством, но для реализации модели топологии посредника также можно использовать Apache Camel (<https://oreil.ly/kwKcS>) или Spring Integration (<https://oreil.ly/GZsnc>).

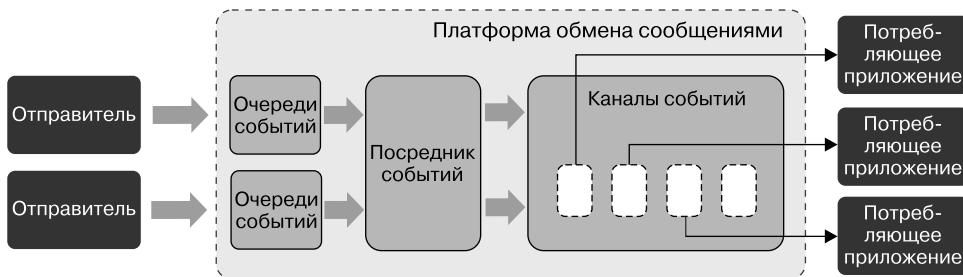


Рис. 5.1. Очередь сообщений гарантирует, что каждое сообщение будет использовано только один раз, удаляя или отмечая его

Топология брокера

Модель топологии брокера тоже работает с сообщениями, но отличается от топологии посредника отсутствием центрального посредника. Вместо этого события передаются легковесному брокеру сообщений, а потом распространяются среди множества потребителей. Эта модель обычно используется в сценариях, где поток событий относительно простой и не требует централизованной координации или оркестрации. Брокеры, которые могут быть централизованными или местными, способны гарантировать, что каждый потребитель получит сообщения из канала или темы в том порядке, в котором они были получены платформой.

Модель брокера, показанная на рис. 5.2, может быть реализована по-разному. Например, она может сразу удалять сообщения после передачи потребителю или использовать политики для сохранения сообщений в течение определенного периода времени. Эта модель полезна для рассылки одних и тех же сообщений нескольким потребителям, например, для передачи информации сразу нескольким системам при обнаружении мошенничества.

Реализация программной архитектуры модели топологии брокера, как и модели топологии посредника, может быть реализована множеством различных способов. Популярный выбор — Apache Kafka (<https://kafka.apache.org/>), а также решения вроде RabbitMQ (<https://www.rabbitmq.com/>) или ActiveMQ (<https://activemq.apache.org/>). Поскольку обработка событий относительно проста и не нужна централизованная

оркестрация и координация событий, эти архитектуры в целом более масштабируемые. В отличие от топологии посредника модель топологии брокера также может использоваться для реализации модели публикации-подписки.

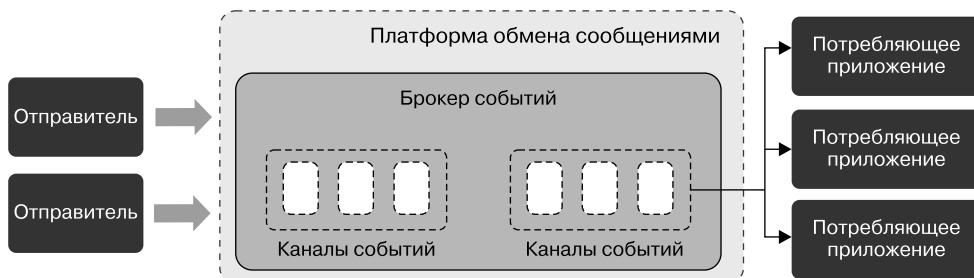


Рис. 5.2. Модель топологии брокера позволяет нескольким потребителям подписываться на одни и те же сообщения

Стили обработки событий

Каждой модели топологии EDA сопутствуют стили обработки событий, зависящие от их сложности и реализуемых вариантов использования. Часто одновременно используется несколько стилей.

Простая обработка событий (simple event processing, SEP) — это самый простой стиль обработки, применяемый к событиям, которые напрямую связаны с конкретными измеримыми изменениями состояния. Никакие сложные рабочие процессы не запускаются. Примером могут быть простые события, которые создаются датчиком, обнаруживающим изменение температуры.

Обработка потока событий (event stream processing, ESP) применяет сложные операции к потокам и нескольким событиям (сообщениям) одновременно, например агрегирование, фильтрация или соединение. Если потоковая обработка на очень высоком уровне генерирует новые потоки только из существующих, то она может не иметь состояния. Эта модель не определяет, хранятся ли данные, но на практике ее часто комбинируют с одной из других моделей (обычно с публикацией-подпиской). Поэтому мы можем рассматривать потоковую обработку как расширение модели публикации-подписки.

Обработка сложных событий (complex event processing, CEP) — очень трудный вариант обработки событий, потому что анализ выполняется для поиска закономерностей, чтобы определить, произошло ли более сложное событие. Принимаемые события могут оцениваться в течение длительного периода времени и даже поступать из нескольких источников одновременно. Сложная обработка событий, как вы скоро узнаете, обычно сочетается с обработкой потока событий.

СЕРВИСНО-ОРИЕНТИРОВАННАЯ И СОБЫТИЙНАЯ АРХИТЕКТУРЫ

Чем SOA отличается от EDA? Или эти архитектуры одинаковы? Обе архитектуры нацелены на высокую гибкость и отстаивают принципы разработки соглашений об обслуживании, возможности обнаружения сервисов, возможности повторного использования сервисов, их абстрагирования и композиции (<https://oreil.ly/nEAJx>). Самое большое отличие в том, что EDA допускает некоторый уровень несогласованности данных, позволяя их копирование, а внутри SOA несогласованность устраняется за счет изоляции сервисов. SOA отстаивает принцип слабой связанности, согласно которому каждый сервис должен быть изолированным и иметь ограниченные зависимости от других общих ресурсов, таких как базы данных, устаревшие приложения или API. SOA также уделяет больше внимания командам, а EDA — событиям.

Реализация событийно-ориентированной архитектуры (EDA) — сложная задача, которая требует гораздо большего, чем просто развертывание платформы потоковой передачи событий вроде Apache Kafka, IBM MQ, Apache Pulsar, RabbitMQ, AWS SNS, AWS Kinesis, Azure Service Bus или Google Cloud Pub/Sub. Комплексная EDA требует множества дополнительных компонентов для создания, сбора, преобразования и использования событий.

Чтобы лучше понять, как работает событийно-ориентированная архитектура, мы углубимся в одну основополагающую технологию — Kafka. Для начала познакомимся с предысторией, затем перейдем к модели API, а потом — к функциям поддержки распределенных приложений.

Введение в Apache Kafka

Платформа Kafka изначально была разработана LinkedIn. У ведущего инженера Джая Крепса (Jay Kreps) и его команды было слишком много потоков данных, пересылаемых в режиме реального времени, и не было хорошей возможности их обрабатывать¹. Они рассмотрели несколько вариантов, но ни один из инструментов не смог справиться с большими объемами данных, генерируемых всеми системами. В результате они создали Kafka — платформу, позволяющую фиксировать события и данные в невероятных масштабах.

Kafka имеет уникальную архитектуру. Она хранит все сообщения в неизменяемых (доступных только для добавления в конец) плоских файлах в распределенной файловой системе. Такой способ отличается от традиционных очередей сообщений, где события хранятся как отдельные записи или файлы и удаляются после их использования. Kafka сохраняет сообщения с параметром, называемым

¹ Джай Крепс предложил интересную концепцию под названием «Архитектура Карпа» (<https://oreil.ly/fjtM5>). Она отличается от других архитектур, потому что вся обработка в ней является событийной.

временем жизни (time to live, TTL), которое обычно составляет семь дней. По истечении этого срока сообщения удаляются.

Kafka — распределенная платформа. Это означает, что данные копируются в кластере из нескольких экземпляров (брокеров). Kafka систематизирует свои данные по темам, напоминающим очереди сообщений. Каждая тема — это группа данных, которые могут быть интересны множеству потребителей. Темы могут содержать все события или быть сжатыми¹ — для каждого уникального ключа содержится только одно недавнее сообщение. Kafka использует разделы для хранения тем. По умолчанию она автоматически балансирует копирование (<https://oreil.ly/M-WOF>). Чем больше разделов, тем лучше масштабируются запись и чтение событий.



Соединяя данные в Kafka, вы должны гарантировать совместимость разбиения на разделы ваших потоков и таблиц (<https://oreil.ly/EinHQ>), то есть входные записи на обеих сторонах соединения должны иметь одинаковые настройки разбиения на разделы. Например, если применяется ключ user_id, то он должен присутствовать в каждом разделе. Я рекомендую использовать одинаковое количество разделов и одинаковую схему ключей. В примере с user_id вы можете использовать десять номеров разделов и определять раздел по последней цифре в user_id. Как видите, управление внешними ключами в разных темах должно осуществляться командой центральной платформы.

Клиент подписывается на тему аналогично тому, как вы подписывались бы на список рассылки. Потребители читают темы, используя *смещение*. Смещение — это текущая позиция в диапазоне событий или сообщений в теме. Смещение может поддерживаться платформой Kafka или самим потребителем (<https://oreil.ly/dnMZk>). После обработки события или сообщения потребитель должен послать подтверждение в Kafka, что приведет к увеличению смещения.

Уникальность Kafka заключается в особенностях взаимодействия с ней. В ней реализовано несколько API (рис. 5.3), с помощью которых можно отправлять, обрабатывать, перенаправлять и потреблять данные. Давайте рассмотрим каждый из них более подробно.

- *Connect API*. Через этот интерфейс можно связать Kafka с внешними системами-источниками и базами данных. Kafka предоставляет множество готовых коннекторов для популярных БД. Они выполняют сериализацию данных и просты в настройке. Connect API может использоваться и производителями, и потребителями событий. При обсуждении чтения и сбора данных о событиях часто используется термин *источник события*, а при обсуждении доставки или сохранения результата в целевом местоназначении — *приемник события*.

¹ Когда сжатие включено, Kafka удаляет все старые записи, если в журнале разделов есть более новая версия с тем же ключом.

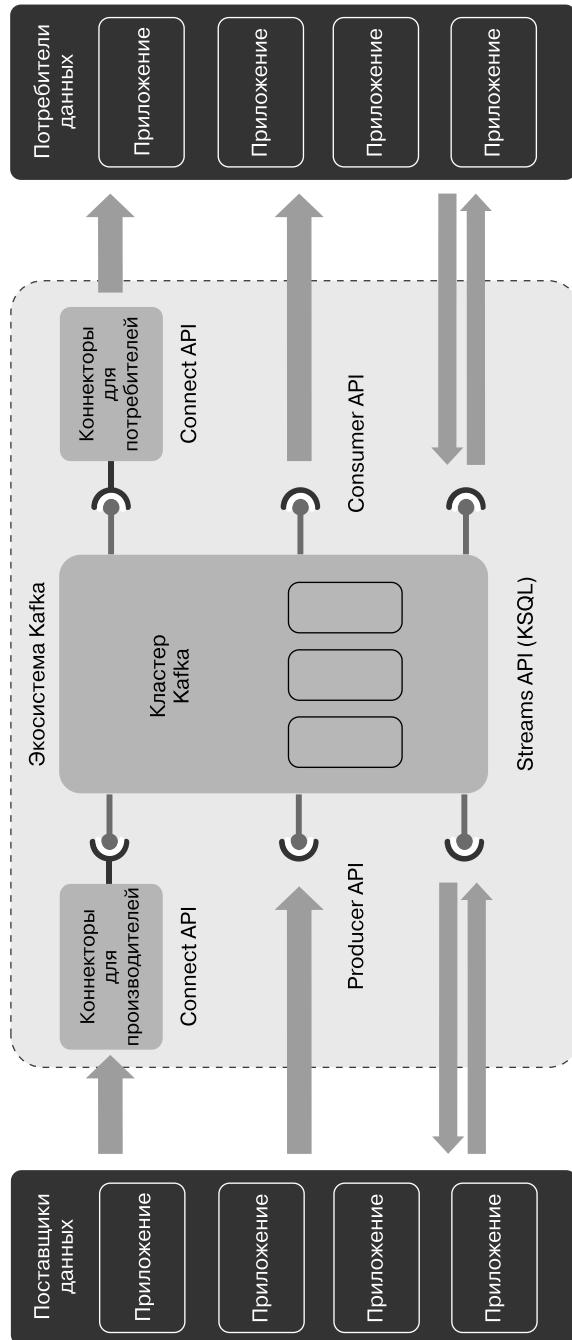


Рис. 5.3. Архитектура API платформы Kafka предоставляет широкий спектр функций

- *Producer API и Consumer API*. Альтернативный вариант — прямая связь через *Producer API* и *Consumer API*. Эта форма связи обычно требует дополнительных функций обработки или кода. Поставщик или производитель событий использует Producer API, а потребитель событий — Consumer API.
- *Streams API* позволяет взаимодействовать и управлять потоками данных внутри Kafka. Поток данных — это непрерывная последовательность событий. Streams API может использоваться как производителем событий, так и потребителем. Здесь производитель может управлять потоком событий или преобразовывать его после доставки. То же относится к потребителю, который может манипулировать потоком событий, прежде чем использовать его.

Streams API позволяет реализовать расширенные шаблоны. Когда данные поступают на платформу или отправляются с нее, их можно перехватывать и управлять событиями в темах. Kafka предоставляет фреймворк — KSQL (<https://ksqldb.io/>), который использует Streams API, для применения (сложных) преобразований. KSQL использует SQL-подобный синтаксис, что объясняет название. Управляя потоками событий, вы также можете фильтровать, агрегировать или объединять темы с другими темами. Поток вновь созданных событий может храниться внутри самой платформы Kafka.

Распределенные события

Еще одна интересная особенность Kafka — поддержка репликации данных. Современные предприятия, как вы узнали из главы 1, используют большое количество приложений в разных местах. Популярность SaaS выросла и многие предприятия стали использовать одну или несколько публичных облачных платформ наряду со своими локальными окружениями. Поэтому возникла необходимость распределения данных между местоположениями.

Kafka предлагает улучшенное решение проблемы распределения данных. Назовем его *зеркаливанием*. С помощью MirrorMaker (https://oreil.ly/_MnSK) (рис. 5.4) можно гарантировать надежную репликацию (копирование) данных в теме кластерами в нескольких физических средах (центры обработки данных или различные облака). Эта репликация не похожа на обычную репликацию между узлами внутри кластера, потому что узлы кластера обычно размещаются в одном физическом месте. Поддержка репликации данных из одной среды в другую делает возможными некоторые интересные шаблоны.

- Создание децентрализованных приложений.
- Объединение данных в реальном времени из нескольких потоков из разных мест: например, создание персонализированного опыта путем объединения данных из интернета, данных о действиях, клиентах и продуктах.

- Рассылка аналитических событий, например, при обнаружении мошенничества или предаварийного или аварийного состояния, в несколько мест.
- Безопасное перемещение данных (состояния) приложения из одного физического места в другое для создания реплики данных только для чтения.
- Взаимодействие и подключение приложений к внешним сторонам, таким как SaaS и внешние поставщики данных.

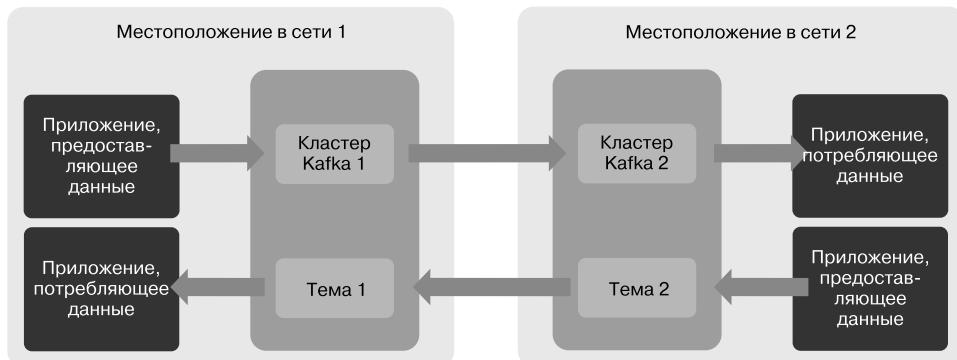


Рис. 5.4. Функция зеркалирования в Kafka позволяет синхронизировать темы между кластерами в нескольких физических средах

Механизм зеркалирования копирует данные асинхронно в обоих направлениях между кластерами. Поскольку данные ставятся в очередь и копируются асинхронно, это позволяет избежать сбоев и повреждения данных из-за нарушений в работе сети. Надежность синхронизации увеличивается также благодаря отказоустойчивой природе кластеров Kafka.

Возможности Apache Kafka

В лице Apache Kafka вы получаете масштабируемую и отказоустойчивую потоковую платформу, предлагающую большое количество возможностей. Платформа позволяет создавать распределенные приложения и обеспечивает:

- *интеграцию данных.* С помощью Kafka и современных потоковых платформ можно напрямую преобразовывать данные или отправлять события внешним механизмам преобразования, например Apache Spark, которые могут выполнять операции ETL самостоятельно. Apache Spark может передавать напрямую преобразованные сообщения обратно в Kafka;
- *сохранение данных.* Kafka можно использовать для хранения огромных объемов данных. Поскольку они не изменяются, можно каждый раз восстанавливать одни и те же предсказуемые результаты. Эти же данные можно

использовать в качестве контрольного журнала или для управления жизненным циклом данных;

- *передачу сообщений или событий.* Kafka действует как простой (сквозной) брокер сообщений для прямого подключения одних приложений к другим. Сюда же входят предупреждения, уведомления, переходы с веб-сайтов, журналы и т. д.;
- *поддержку микросервисов.* Kafka часто используется в архитектурах микросервисов и способствует синхронизации данных между микросервисами, источниками событий, источниками команд и CQRS;
- *потоковую передачу и анализ в режиме реального времени.* Брокеры Kafka могут использоваться для анализа потоков данных в режиме реального времени с их функциями временных рядов и возможностями преобразования;
- *репликация данных.* Кластеры Kafka можно настроить для надежного копирования тем из одного кластера Kafka в другой. Эти кластеры могут размещаться в разных физических местах.

Недаром Kafka используется во многих крупных компаниях и на предприятиях как ключевой компонент архитектуры EDA. Kafka — это сложная платформа распространения с расширенными возможностями интеграции и API, пользующаяся широкой поддержкой сообщества.

Потоковая архитектура

Вернемся к нашей потоковой архитектуре и определим, как событийно-ориентированные характеристики могут дополнять архитектуры RDS и API. Благодаря потоковой передаче событий можно создавать приложения, действующие в режиме реального времени, преобразовывать данные по мере их поступления и передавать их другим приложениям и системам. Реактивный характер событий, распределенные возможности и способность транспортировки баз данных приложений делают потоковую передачу важной частью вашей архитектуры.

Событийно-ориентированная архитектура состоит из трех основных архитектурных блоков: производителей событий, потребителей событий и платформ. Производители событий публикуют свои события и будут обсуждаться в следующем разделе. Вслед за ними мы рассмотрим потребителей и, наконец, платформы.

Производители событий

На стороне поставщиков событий (часто называемых *производителями событий* или *издателями*) многие исходные системы не передают данные и не генерируют событий сами по себе. Очень часто нужны еще компоненты для сбора данных из источников и преобразования протоколов или дополнительной интеграции,

например, преобразования сообщений в нужный формат, фильтрации, агрегирования и обогащения. Взглянем на некоторые сценарии и варианты.

- *Внешние интерфейсы и события пользовательского интерфейса.* Приложения могут быть спроектированы так, что события генерируются внешними и пользовательскими интерфейсами: например, события просмотра веб-страниц и добавления продуктов в корзину в браузере могут собираться напрямую¹. Если приложение использует трехуровневую программную архитектуру, события генерируются уровнем представления (рис. 5.5).



Рис. 5.5. События можно собирать или генерировать со всех трех уровней приложений

- *События приложений.* События и уведомления могут отправляться самими приложениями. В этом случае за создание таких событий отвечает уровень бизнес-логики. Возьмем для примера приложение CRM, написанное на Java. Когда учетная запись пользователя блокируется после трех неудачных попыток входа в систему, система автоматически генерирует событие. Для генерации или отправки событий в некоторых случаях нужны дополнительные клиентские библиотеки. Пример — приложение веб-сервера Node.js, которое после получения заказов от мобильных приложений отправляет данные прямо в Kafka с помощью Producer API и библиотеки Kafka-node.
- *Чтение базы данных приложения.* Другой шаблон генерации событий — это получение изменений в БД приложения через чтение уровня хранения данных². Это можно организовать путем опроса или с помощью методов CDC (change data capture — «захват изменений в данных»), компонентов на основе журналов или прямого чтения БД. Такая модель приема событий, несущих

¹ События, генерируемые внешними и пользовательскими интерфейсами, являются односторонними (запустил и забыл), потому что обычно от получателей сообщений не требуется давать ответ.

² Эта форма состояния также известна как состояние ресурса или текущее состояние ресурса на сервере в любой момент времени. Некоторые инженеры различают состояние приложения на стороне клиента и состояние ресурса на стороне сервера.

данные, также известна как *передача состояния с переносом событий*. Во многих случаях требуется также добавлять дополнительные компоненты и прикладывать некоторые усилия.

Давайте рассмотрим несколько методов передачи состояния с помощью событий.

Распространенный метод (**❶** на рис. 5.6) – использование коннекторов для БД и запуск опроса (<https://oreil.ly/3lxkV>): таблицы базы данных сопоставляются с очередями сообщений или темами через повторное выполнение запросов (обычно с помощью Open Database Connectivity или Java Database Connectivity). Каждый раз, когда в таблицу добавляется новая или изменяется существующая запись, она извлекается и передается либо в очередь сообщений, либо в тему. Метод на основе опроса обычно проще в настройке, но он может создавать более высокие нагрузки на базу данных из-за постоянного сканирования. Еще один недостаток опроса – необходимость использования обходных путей. Предварительным условием может быть дополнительный столбец `updated_at`, необходимый для определения новых записей.

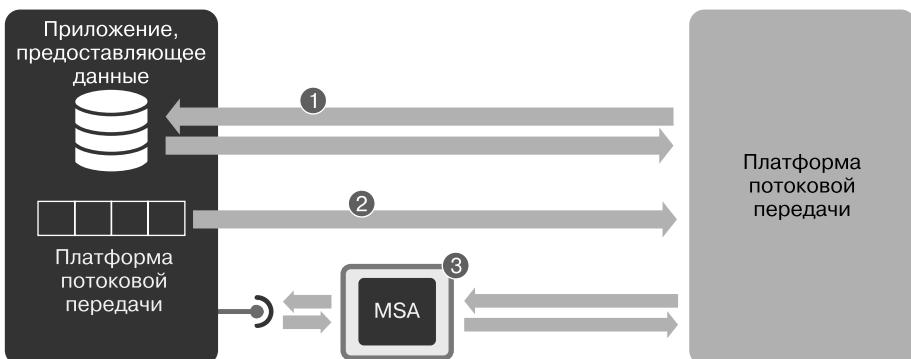


Рис. 5.6. События для передачи состояния можно создавать по-разному

Другой метод **❷** передачи состояния – метод CDC – основан на использовании файлов журналов. Он определяет изменения в базе данных, читая журнал транзакций. Инструменты на основе CDC реже используют базу данных, потому что не запрашивают ее напрямую, но во многих случаях требуются дополнительные коммерческие инструменты. Примеры: Attunity (<https://oreil.ly/vsB1v>), Precisely (<https://oreil.ly/XyF8d>), StreamSets (<https://streamsets.com/>), SQData (<https://www.sqdata.com/>), IBM InfoSphere CDC (<https://oreil.ly/wossx>), Oracle GoldenGate (<https://oreil.ly/99tcQ>), Debezium (<https://debezium.io/>) и CosmosDB Change Feed (<https://oreil.ly/DDvGc>).

Еще один метод (**❸** на рис. 5.6) – самостоятельно написать инфраструктуру для сканирования БД или генерации событий. В этом случае для создания событий

используются не (комерческие) коннекторы или инструменты на основе журналов, а небольшие прикладные компоненты, которые читают базы данных или API или постоянно прослушивают их. Примером может служить HTTP-прокси, прослушивающий каналы связи. В наши дни популярным подходом является использование микросервисов.

В зависимости от структуры исходной системы и выбранного метода сбора данных могут потребоваться дополнительные упрощенные преобразования. Если структуры данных сложны для понимания или их необходимо объединить с другими данными, тогда можете использовать удобные инструменты для разработчиков, такие как Apache NiFi (<https://nifi.apache.org/>) и Apache Flume (<https://flume.apache.org/>), позволяющие создавать оптимизированные потоки для эффективного сбора, агрегирования и преобразования данных из потоковой платформы. В качестве альтернативы можно организовать преобразования на самой потоковой платформе. Распространенный сценарий — повторная обработка исходных и временных данных с использованием правил очистки, преобразования или агрегирования. Мы обсудим это в подразделе «Бизнес-потоки» на с. 173.

Потребители событий

Теперь переключим внимание на потребителя события и оценим существующие шаблоны проектирования. Разнообразие вариантов на стороне потребителя обусловлено требованиями сценариев использования и выбранной технологией. Рассмотрим несколько сценариев и обсудим возможности.

Самый простой вариант использования очереди сообщений или брокера сообщений — отделить одного поставщика от одного потребителя с помощью брокера сообщений. Производитель публикует сообщения в канал или тему единственного раздела, а потребитель получает сообщения из раздела. В Kafka, например, для экспорта событий из потоковой платформы в БД можно использовать коннектор приемника, использующий простые операторы INSERT¹.

Простая вариация примера выше — когда несколько потребителей используют один поток событий. В этом случае каждый потребитель поддерживает свое смещение текущего смещение сообщения². Этот шаблон известен как *разветвленный обмен*.

Труднее будет расширить операционную систему с помощью современных функций приложения. Чтобы преодолеть этот недостаток, в облаке развертывается компонент приложения FaaS (см. подраздел «Функции» на с. 139). Этот компо-

¹ Коннектор приемника — это термин, используемый для компонентов, которые доставляют данные из тем Kafka в другие системы.

² Для Kafka эти потребители должны находиться в разных группах потребителей.

мент работает в дополнительном месте и вызывается через события. Функция обрабатывает сообщение, сохраняет результат или выполняет поиск в другой БД, потом отправляет результат обратно в операционную систему. Такой шаблон еще известен как *шаблон Strangler* («*Душитель*») (см. врезку «Микросервисы и бессерверные архитектурные шаблоны» на с. 139). Поддерживающая функция или компонент приложения в этом примере может рассматриваться как микросервис.

СРАВНЕНИЕ ОБРАБОТКИ ПОТОКОВ СОБЫТИЙ И СЛОЖНЫХ СОБЫТИЙ

Различия обработки потока событий и сложных событий хорошо объяснил Сринат Перера (Srinath Perera) на Quora (<https://oreil.ly/YkNvwa>). Обработка сложных событий (complex event processing, CEP) считается подмножеством (рис. 5.7) обработки потока событий. Ее можно использовать в зависимости от задачи и варианта использования, с которым вы имеете дело.



Рис. 5.7. Обработка сложных событий — это подмножество обработки потока событий

Для анализа одной последовательности событий, упорядоченных по времени, или создания графа обработки используйте приемы обработки потоков событий. Для параллельного анализа событий при помощи языка запросов высокого уровня используйте СЕР. Следуя этим высокоуровневым различиям, вы выявите два типа механизмов: механизмы обработки потоков событий и механизмы обработки сложных событий. Первый больше ориентирован на такие операции с данными, как фильтрация, проверка порядка и т. д. Второй делает больший упор на сложный анализ, такой как сравнение временных окон, выявление корреляции между различными событиями и т. д.

Но постепенно различия сглаживаются. Механизмы обработки потоков событий заимствуют возможности СЕР. Хорошим примером является платформа Apache Kafka, добавившая поддержку KSQL для анализа событий.

Расширенный сценарий может заключаться в использовании механизма обработки потока или аналитической платформы, такой как Apache Storm или Apache Spark, для анализа и поиска закономерностей в общем потоке событий нескольких поставщиков. Появление определенных событий в определенном

временном окне или совпадений в наборе данных может вызвать другое событие для автоматического выполнения действий или информирования других систем. Этот шаблон запроса и анализа данных перед сохранением в базе данных также известен как *обработка сложных событий*. Многие платформы потоковой передачи позволяют выполнять аналитику, поддерживая нескольких оконных функций¹, таких как переворачивающееся окно, всплывающее окно, скользящее окно и окно сеанса.

Другой способ использования платформы потоковой передачи — не реагировать на события и не прослушивать их, а передавать состояние приложения. Передача состояния отличается от предыдущих примеров, поскольку основная ее цель — поддерживать синхронизацию данных, а не принимать меры в отношении событий. Обе цели, передача состояния приложения или базы данных и реагирование на события, можно объединить. Шаблон передачи состояния с переносом событий часто комбинируется с CQRS, что будет рассмотрено в следующем разделе.

Платформа событий

Суть событийно-ориентированной архитектуры уловить еще сложнее. В основе лежат брокеры событий для управления взаимодействием между поставщиками и потребителями. Хотя основной подход заключается в открытии соединений, получении сообщений и их отправке нескольким потребителям, его отличает именно отслеживание состояния. Многие потоковые платформы хранят данные и предотвращают их потерю с помощью распределенных возможностей. Можно запросить базу данных или журнал, доступный только для добавления, и в этом проявляется сходство с хранилищем пар «ключ — значение». По мере поступления потоков данных платформа может преобразовывать, комбинировать, фильтровать или агрегировать любые из них. Кроме того, из этих потоков можно создавать новые параллельные потоки. Наконец, некоторые из этих платформ предлагают сложные функции обработки событий, поэтому вы можете проводить анализ на стороне потребителя или на платформе потоковой передачи.

Существует множество вариантов создания ядра EDA. Один из них — самостоятельно построить событийно-ориентированную архитектуру с использованием различных компонентов и технологий. Для хранения данных можно использовать современные базы данных с интерфейсами REST. Микросервисы или инструменты, такие как Apache Flink (<https://oreil.ly/tM15Z>), Apache NiFi или Apache Spark, можно применять для решения задач интеграции данных и аналитики. Традиционные очереди сообщений можно настроить с помощью ActiveMQ,

¹ Чтобы узнать больше об аналитических возможностях, посетите Kafka (<https://oreil.ly/T3k2n>) и Microsoft (<https://oreil.ly/BV-gN>).

ZeroMQ или других решений. RabbitMQ поддерживает создание очередей, но также имеет модель брокера, с помощью которой направляет сообщения в разные очереди с ключами привязки и маршрутизации. Еще один вариант — использовать Apache Kafka с ее мощными возможностями интеграции и широкой поддержкой сообщества.

Основные поставщики облачных услуг предлагают аналогичные возможности. В AWS, например, доступны DynamoDB (<https://oreil.ly/xg5uz>) (база данных), Simple Queue Service (SQS) (https://oreil.ly/tkuV_) и Kinesis (обработчик событий) (<https://oreil.ly/yMwsc>). В Azure аналогичные возможности предлагаются в виде Event Grid (<https://oreil.ly/jqMn8>), Event Hubs (<https://oreil.ly/-SvCu>), Service Bus (<https://oreil.ly/Hutqp>), CosmosDB (<https://oreil.ly/UgjkO>) и Queue Storage (<https://oreil.ly/wV2y3>). В Google поддерживаются Dataflow (<https://oreil.ly/TFonT>) и Apache Beam (<https://oreil.ly/ZxnN0>).

Используя эти платформы или инструменты как основу EDA, можно распределять и хранить события, а еще сохранять полную историю взаимодействий приложений и реализовать некоторые продвинутые шаблоны проектирования. Один из них — *событийно-ориентированная хореография сервисов*: распространение обновлений и команд в асинхронном режиме, что позволяет нескольким приложениям принимать и обрабатывать эти обновления и команды. Мы обсуждали этот шаблон в главе 4. Два других шаблона, которые можно использовать при увеличении периода хранения (<https://oreil.ly/zcvCS>), — источник событий и источник команд.

Источники событий и команд

При использовании шаблона «источник событий» (<https://oreil.ly/Tath1>) все изменения в состоянии приложения или БД сохраняются как последовательность событий, обычно в неизменяемом журнале или хранилище, доступном только для добавления. Реализация этого шаблона дает:

- контрольный журнал, позволяющий просматривать все события, имевшие место и использовавшиеся для обновления хранилища данных приложения;
- возможность восстановить любое предыдущее состояние системы. Можно еще использовать дополнительные события, чтобы обратить или отменить изменения при восстановлении текущего состояния;
- возможность возвращаться назад, воспроизводить и события и манипулировать ими. Так можно определять закономерности в поведении пользователей или другие полезные особенности;
- возможность агрегировать и материализовать события. Это позволяет идеально форматировать данные для требуемых операций запроса.



Источники событий иногда путают с CRUD (<https://oreil.ly/pIgvt>). Источники событий в целом фокусируются на семантике, в то время как CRUD — на технических возможностях создания, чтения, обновления и удаления.

Получив все события, хранящиеся последовательно, вы также можете полностью перестроить хранимые данные приложения в других местах или скопировать их из одного места в другое. Этот шаблон можно использовать для практического применения CQRS: создания материализованных представлений, основанных на событиях. Это означает, что потенциально можно создать *несколько распределенных хранилищ данных только для чтения* (рис. 5.8) в разных местах из одного и того же потока событий. Все эти хранилища постоянно снабжаются свежими данными.

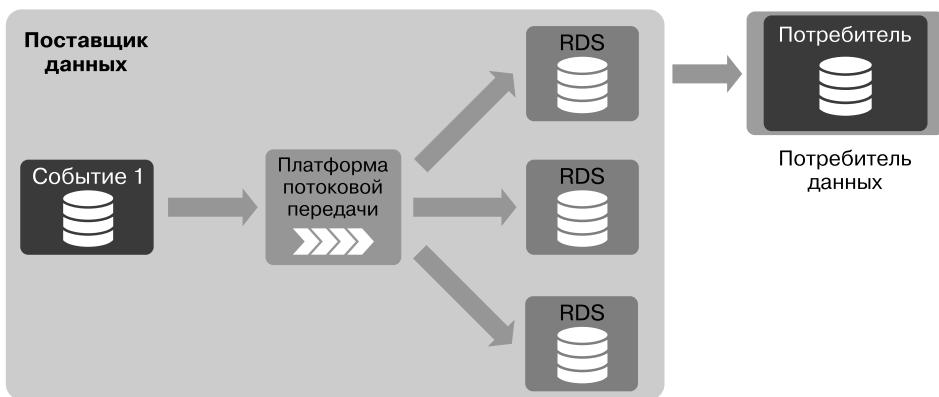


Рис. 5.8. С помощью источника событий можно синхронизировать данные сразу с несколькими хранилищами данных только для чтения

Другой вариант — вернуться назад и воспроизвести все исходные события, как если бы вы начинали заново. Во время воспроизведения можно исправлять, изменять, обогащать события или присоединяться к потоку. Например, обнаружив ошибку, можно применить ретроспективное исправление, а затем воспроизвести все события как новые. Прием возврата назад также можно использовать для объединения различных потоков событий. Например, можно перемотать назад серию событий, используя справочную таблицу.

Наконец, благодаря источнику событий предметным областям не нужно беспокоиться о координации с другими областями. Приложения, генерирующие события, отделены от приложений, подписывающихся на события. Это означает,

что предметные области могут работать с одним и тем же потоком событий или генерировать эти события, а также делать что угодно с данными, не нарушая и не изменяя данные других областей.

ХРАНИЛИЩА СОСТОЯНИЙ

Источники событий и Streams API могут быть объединены для упрощения CQRS за счет создания хранилищ состояний (<https://oreil.ly/LPv68>). Хранилища состояний можно сравнить с интегрированными децентрализованными представлениями, которые постоянно заполняются новыми событиями. Они могут находиться как в памяти, так и в долговременных хранилищах состояний внутри приложения.

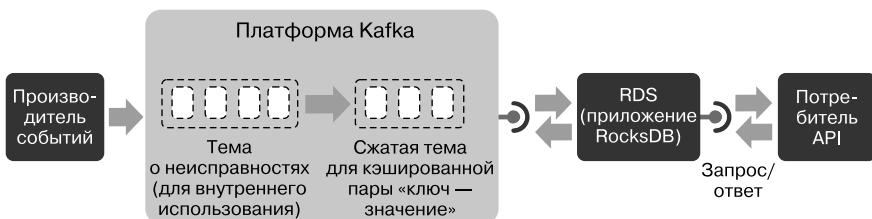


Рис. 5.9. Kafka Streams позволяет обрабатывать потоки с отслеживанием состояния. Данные о состоянии хранятся в «хранилищах состояний» — простых хранилищах пар «ключ — значение» в БД, например, RocksDB. К этим хранилищам состояний можно посыпать запросы, что делает их подходящими для применения шаблона проектирования CQRS

Примером (рис. 5.9) может служить заполнение постоянного хранилища пар «ключ — значение», например RocksDB (<https://rocksdb.org/>), информацией из конечной точки REST. Вместо обращения к операционным системам вы обращаетесь к хранилищу пар «ключ — значение» после копирования и доставки данных ближе к потребителю. Эти хранилища состояний, которые играют роль RDS, могут даже находиться в разных местах сети. Компонент ksqlDB (<https://oreil.ly/c0dvv>) платформы Kafka работает аналогично, позволяя приложениям искать значения в вычисленных таблицах в KSQL.

Таким образом, объединение источников событий с хранилищами состояний позволяет запрашивать данные распределенным образом.

Еще один шаблон, который можно реализовать, — *источник команд*. Он работает в направлении, противоположном источнику событий, и основан на перехвате команд. Прежде чем событие, несущее команду, попадет в обрабатывающую систему, оно помещается в очередь или тему и уже оттуда доставляется в целевую систему. Идея сохранения всех команд заключается в возможности предвидеть плохие ситуации. Если по какой-либо причине система будет повреждена, вы сможете перемотать команды и начать обработку заново.

Модель управления

Давайте сделаем шаг назад и рассмотрим модель управления поставщиками событий и потребителями. Архитектура потоковой передачи следует той же модели, что и архитектуры RDS и API. Все потоковые каналы вроде тем и очередей должны принадлежать поставщикам событий (поставщикам данных). Когда один и тот же кластер используется как для частной, так и для публичной связи, ситуация может быстро усложниться, потому что требуется предотвратить возможность подписки потребителей из одних областей на частные каналы связи из других. Для этого я рекомендую строго разделять (рис. 5.10) частные и публичные потоки¹. Вы уже видели эту схему классификации для API в главе 4, где использовались технические и бизнес-API.

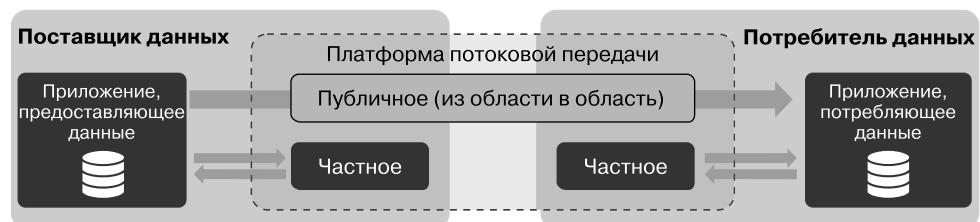


Рис. 5.10. Частные события должны быть отделены от публичных, если для передачи событий используется один кластер

Частные и публичные потоки можно разделить с помощью нескольких методов. Соглашение об именах — один из них: встраивание в имя темы или очереди названия области. Другой, более строгий метод — применение авторизации. Сертификаты, которые могут быть прочитаны другими областями — еще более строгая форма защиты потоков. Также для частных и публичных потоков должны устанавливаться разные политики.

События, используемые для частной связи (например, между двумя компонентами приложения), могут быть техническими; от них не требуется иметь какое-то определенное значение для бизнес-логики. Потоки для таких событий называются *частными потоками* и считаются частью внутренней логики приложения или области. Это означает, что они могут потребляться только в области, производящей их, и, следовательно, не могут потребляться напрямую другими областями. События из частных потоков могут передаваться в бизнес-потоки. В отношении метаданных можно быть не такими строгими.

¹ Осторожно: не путайте их с частными темами, которые Kafka Streams использует для создания хранилищ состояний.

Например, частные потоки могут перечисляться в реестре очередей или тем, но сопровождаться менее подробным описанием. *Реестр* – это центральное место для регистрации всех очередей и тем, включая владение, управление версиями, метаданные схемы и управление жизненным циклом.

Публичные потоки, напротив, предназначены для потребления другими областями. Публичные потоки, также называемые *бизнес-потоками*, готовы к использованию в бизнесе и оптимизированы для чтения. Они отделены от внутренней логики областей. Это означает, что они остаются совместимыми и перечислены в реестре с более подробной информацией для потребителей данных. Они следуют всем принципам из главы 2.

Бизнес-потоки

Когда предметные области хотят обмениваться данными или доставлять события в другие области, они должны следовать той же политике, что и архитектуры RDS и API. Задача создания значимых бизнес-потоков может вынудить поставщиков переписывать события. Мы не можем ожидать, что в окружениях с сильно нормализованными БД потребители событий присоединятся и перестроют сложную логику базы данных и приложения. О вариантах, к которым это может привести, вы узнаете больше в следующих разделах.

Внутри границ приложения

Для устаревших приложений, таких как мейнфреймы, удобнее выполнять преобразование внутри самого приложения (границ области). Суть заключается в том, чтобы сначала преобразовать сложные внутренние структуры данных в структуру, имеющую большее бизнес-значение. Преобразования могут включать соединение таблиц, фильтрацию, преобразования типов данных, конкатенацию и т. д. Конечный результат – денормализованная таблица результатов (рис. 5.11), которая находится в БД приложения и служит источником данных для потока.

Преимущество такого подхода в том, что потребители не обременены сложной логикой. Если объект клиента изменяется, но данные распределены по шести различным техническим событиям, потребителям будет очень сложно увидеть ситуацию в целом. Чтобы было проще сопоставить изменения с общей картиной, нужно предварительно интегрировать все данные.

Можно также преобразовать данные в более подходящий формат с помощью инструментов CDC или дополнительных очередей сообщений, таких как IBM MQ. В этом случае разработчикам приложений не придется изучать слишком много нового. Преобразования происходят в границах области применения.

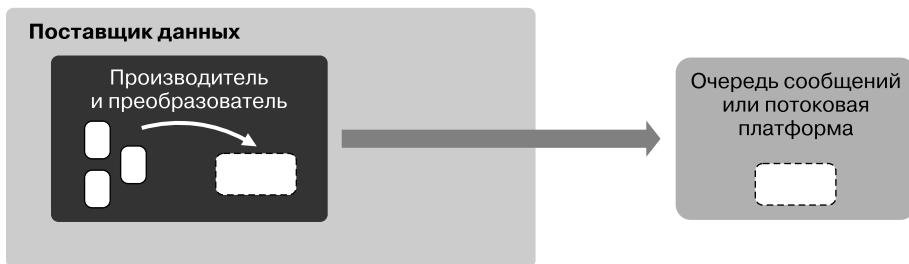


Рис. 5.11. Преобразованием частных потоков в бизнес-потоки управляет поставщик событий в самом приложении

Использование сторонних инструментов

При работе над приложениями, имеющими структуры данных, сложные для настройки или для доступа, можно воспользоваться сторонними инструментами (рис. 5.12) создания бизнес-событий. Преобразование исходных технических данных в события для публичных потоков происходит за пределами приложения, но в границах той же предметной области.

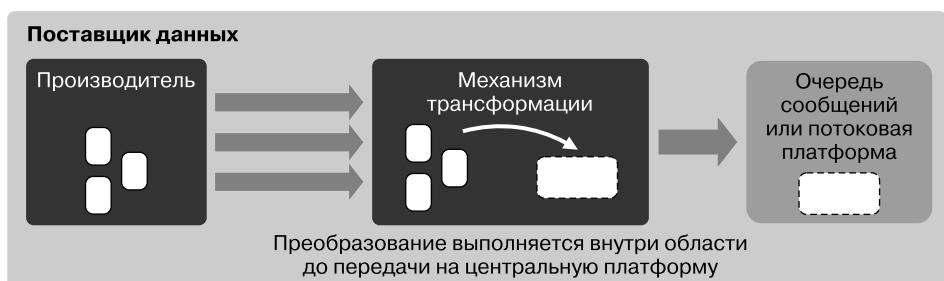


Рис. 5.12. Такие инструменты, как Apache NiFi и Apache Flume, могут идеально разместиться между приложением и потоковой платформой

Этот подход сочетает преобразование сообщений с обработкой поиска и дополнительными услугами. Вызвав внешний API, можно дополнить событие новыми данными, — например, справочными данными, данными из операционных систем или аналитическими данными. После обогащения данные будут готовы к отправке на центральную потоковую платформу для передачи другим предметным областям.

Если вы используете простой брокер сообщений без возможностей преобразования, то дополнительный шаг преобразования неизбежен. Если производители событий будут выдавать только технические сообщения, то потребители

столкнутся со сложными структурами, которые им нужно будет изменить. Потребители всегда должны преобразовывать дату рождения, которая выражена числом, а не настоящей датой. Еще есть риск, обусловленный созданием тесной связи: изменения в событиях на стороне производителей немедленно повлияют на потребителей и потребуют изменения логики интеграции. Решить эту проблему можно отделением с использованием сторонних инструментов или дополнительных компонентов приложения.

Использование подходов после добавления

Еще один подход к отделению — преобразование технических данных после их передачи на центральную платформу. Например, Kafka KSQL позволяет преобразовывать данные из одной структуры в другую в реальном времени на платформе (рис. 5.13). При таком подходе к доставке событий важно не смешивать события из разных потоков и четко разделять частные и публичные потоки.

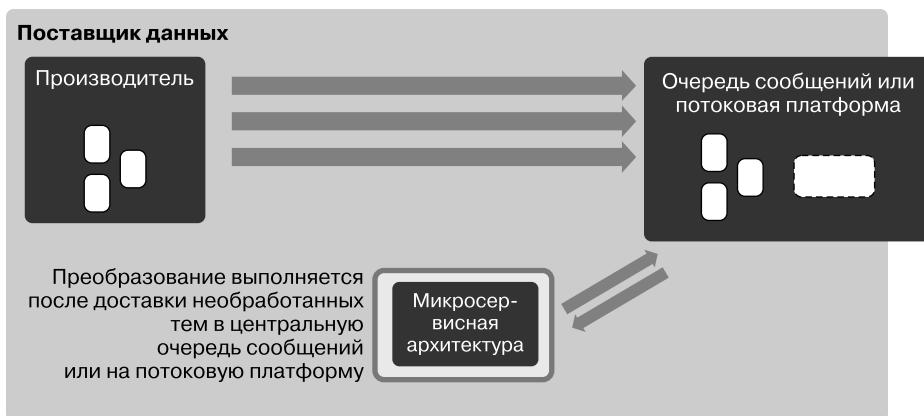


Рис. 5.13. Преобразование событий выполняется после доставки технических сообщений на центральную платформу

Преобразование технических данных в менее нормализованные события для публичных потоков также может происходить за пределами центральной платформы. Популярный метод в микросервисной архитектуре — сначала загрузить технические данные на платформу, использовать их с помощью микросервисов, преобразовать или улучшить и сразу же отправить обратно на центральную платформу.

Такой подход можно сочетать с использованием монолитной системы: здесь она делает копию своих данных доступной через центральную потоковую платформу. Микросервис потребляет данные, выполняет логику преобразования

и отправляет результат обратно на центральную платформу. Все это может происходить в пределах области поставщика данных, если ответственность четко определена.

Шаблоны потребления потоковой передачи

На стороне потребления все еще сложнее, ведь потребности более разнообразны и зависят от варианта использования. События, которые потребляются и преобразуются в границах (потребляющей) области, могут использовать платформу потоковой передачи, а также дополнительные фреймворки и компоненты, предоставляемые самой областью. Не забывайте отмечать потоки как частные или публичные, потому что потребители событий могут стать поставщиками, но при этом решать, оставить ли новые потоки событий только для себя, не открывая их другим областям. Во всех случаях область-потребитель устанавливает требования и берет на себя ответственность за этап преобразования на стороне потребителя. На рис. 5.14 показаны некоторые из возможных моделей потребления. Давайте обсудим варианты обработки потоковых данных потребителями событий.

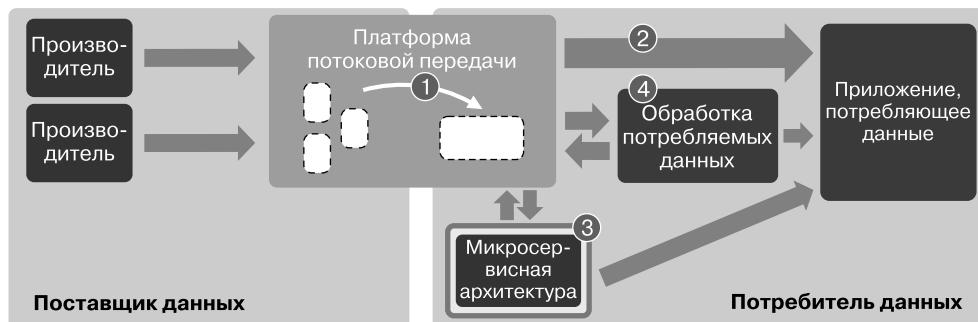


Рис. 5.14. Потребители событий могут выбирать из множества различных шаблонов

Использование возможностей платформы

Первая модель потребления напрямую использует центральную платформу для преобразования событий. В Apache Kafka, например, можно разрешить потребителям событий использовать возможности распределенной обработки для преобразования событий из формата производителя в формат потребителя, в частности, с помощью KSQL. События от производителей можно преобразовать в другой поток, хранимый внутри платформы (1 на рис. 5.14),

или преобразовывать и немедленно пересыпать **❷** приложению-потребителю в желаемом формате.

Микросервисы и фреймворки событий

Другой шаблон потребления — использование микросервисов **❸** для преобразования и обогащения событий в соответствии с требованиями потребителя. Например, Apache Kafka Streams (<https://oreil.ly/xVLJf>) имеет возможности, позволяющие микросервисам взаимодействовать с потоком данных (обогащая, анализируя или преобразовывая его). Для этой цели также можно использовать другие платформы, такие как Spring Cloud Stream (<https://oreil.ly/mG9yP>), Akka (<https://akka.io/>) или Axon (<https://oreil.ly/GYmey>).

Во всех случаях можно есть возможность передавать преобразованные данные обратно на центральную платформу или напрямую потребителю. Приложение-потребитель тоже может быть микросервисом с собственной БД, запоминающим текущее смещение (последнее успешно полученное сообщение). Если область напрямую передает события обратно на центральную платформу, то они могут стать доступными и для других областей. Очень важна правильная регистрация тем и очередей сообщений, иначе частные каналы могут рассматриваться как публичные события.

Сервисы потоковой аналитики

Для аналитических сценариев использования (*обработка сложных событий*) можно задействовать дополнительные потоковые компоненты (**❹** на рис. 5.14). Они могут различаться и включать анализ в реальном времени, визуализацию, сохранение событий в базах данных, вызов дополнительных API, оркестрацию обработки событий и многое другое. Apache Samza, Apache Spark Streaming, Apache Beam и Apache Storm — это лишь некоторые из популярных фреймворков с открытым исходным кодом. Основные поставщики облачных услуг предлагают аналогичные возможности. Amazon предлагает Kinesis Analytics (<https://oreil.ly/TD9Cb>), Microsoft — Azure Stream Analytics (<https://oreil.ly/zAVm5>), a Google Cloud — Dataflow (<https://oreil.ly/3bPIJ>).

Некоторые из этих компонентов имеют распределенную архитектуру обработки. Например, Apache Spark Streaming обладает высокой масштабируемостью и может одновременно обрабатывать и сохранять огромное количество событий. И снова у вас есть возможность передавать события, созданные в результате агрегирования, объединения, преобразования и анализа, обратно на центральную платформу. Если нужно смешать разные шаблоны, может быть полезно использовать центральную платформу как ядро, маршрутизирующее события между разными компонентами.

Маршрутизация событий в публичном облаке

Еще хочу рассказать вам о маршрутизации (пересылке) событий в общедоступном облаке с помощью таких инструментов, как AWS Kinesis и Azure Event Hubs (<https://oreil.ly/ZaRye>).

Хотя эти возможности могут перекрываться центральной платформой потоковой передачи, такой как Kafka, все равно будет полезно использовать подобный дополнительный компонент (рис. 5.15, справа) внутри области. Преимущество здесь в слабой связи, ведь области могут доставлять события одновременно в несколько БД в своей (облачной) среде, независимо от центральной команды разработчиков. Кроме того, замена приложений не требует консультации с центральной командой.

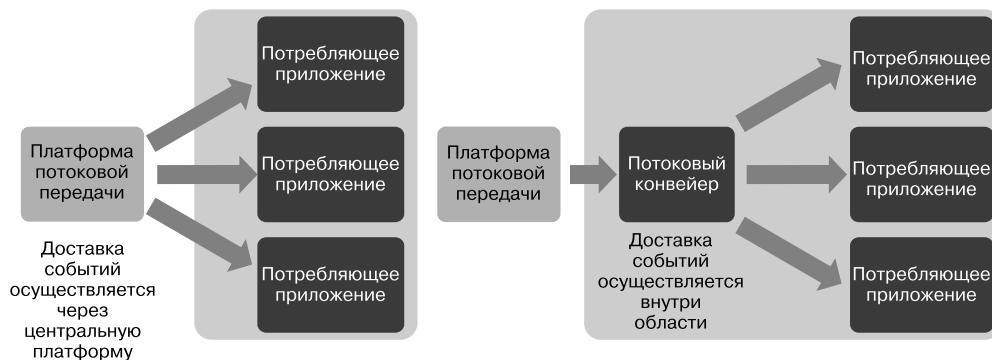


Рис. 5.15. События могут поступать непосредственно с центральной платформы (слева) или через дополнительный компонент в области (справа)

Передача состояния с помощью событий

Я много говорил о передаче событий из одной области в другую с использованием возможностей централизованной потоковой передачи. Но есть еще один шаблон, который предполагает передачу состояния приложения внутри области с использованием частных потоков. Обычно платформы потоковой передачи используются для отделения разных областей друг от друга, но в потоковой архитектуре они также могут использоваться для передачи состояний между приложениями внутри области.

Для примера возьмем операционную систему, использующую средства аналитики, предлагаемые облачным поставщиком. Асинхронная связь обеспечивает устойчивость к сбоям в сети, поэтому идеальной формой обмена данными было бы использование функций распределенной репликации данных центральной потоковой платформы.

ДЕТАЛИЗАЦИЯ СОБЫТИЙ

При разработке событий и их детализации учитывайте возможность передачи состояний и уведомлений. Для передачи состояния необходима поддержка публикации событий со всем агрегированным состоянием. В этой модели получателям передаются с событиями все подробности. Поэтому потребителям приходится приложить дополнительные усилия, чтобы выяснить, что именно изменилось. Этот подход имеет общие черты с ресурсо-ориентированной архитектурой и REST-интерфейсами.

Для уведомлений о событиях используется иная модель — распространяется только то, что было изменено. Это может быть новый идентификатор клиента, обновленный пароль или изменение адреса. Преимущество такого подхода в том, что объем данных в событии небольшой и не они нагружают сеть. Недостаток же в том, что потоков и разных событий быстро становится больше.

Я рекомендую облегчить этот частный канал связи и в то же время предложить области доставлять многоразовые события, чтобы они также были доступны другим потребителям. Альтернативный вариант — сделать частными все потоки, принадлежащие одному ограниченному контексту, и ограничить их доступность для других областей.

Роли RDS

Может ли потоковая платформа использоваться как БД и играть роль хранилища только для чтения, когда требуется хранить данные в течение длительного времени? Абсолютно. Благодаря источникам событий, хранилищам состояний и возможности возврата назад можно создавать интегрированные RDS, которые постоянно заполняются свежими данными. Потребители могут сгенерировать копию данных, пользуясь методом возврата назад. Все события читаются заново, от первого до последнего. После возврата назад пользователь может применить потоковую передачу для поддержки своей копии данных в актуальном состоянии в режиме реального времени. Такой метод можно упростить с помощью CDC, прочитав все изменения из журналов транзакций приложения.

Если потребуется более длительное хранение данных, то стоит определить, при каких условиях потоки могут дольше или бесконечно (постоянно) хранить данные. Опасность заключается в том, что при бесконтрольном увеличении сроков хранения может закончиться место на диске. Это может вызвать сбой в работе платформы. Поэтому имейте в виду, что контроль необходим. Один из принципов, который можно установить, — разрешить сохранение только сообщений с уникальными парами «ключ — значение». В качестве альтернативы можно выгружать сообщения в архитектуру RDS, используя менее дорогие варианты хранения.

Использование потоковой передачи для заполнения RDS

Хранилища данных только для чтения могут создаваться из потоков. Данные в этой ситуации сначала загружаются, при необходимости преобразуются, после чего передаются с потоковой платформы во вновь созданные БД. Возможности распределенной платформы, такие как зеркалирование, также могут помогать заполнять RDS в разных физических местах. Во всех случаях важно придерживаться принципов, обсуждаемых в главе 2, — оптимизирование для чтения, управление владением, регистрация метаданных и т. д.

Создавать хранилища RDS можно и на основе частных, и на основе публичных потоков. Частные потоки, например, можно передавать на вход дополнительного этапа преобразования, выполняемого на платформе потоковой передачи, а его выход и публичные потоки можно использовать для создания RDS. События можно хранить в платформе потоковой передачи или создать отдельную БД за ее пределами. Использование отдельной базы данных позволит оптимизировать ее для конкретных случаев использования. Например, если вы решите оптимизировать онлайн-канал с помощью RDS, то для этого лучше подойдет БД типа «ключ — значение», а для бизнес-аналитики лучше использовать RDBM.

Элементы управления и политики для управления доменами

Может ли центральная потоковая платформа использоваться в качестве серверной части приложения? Например, можно ли использовать Kafka как хранилище данных? Теоретически это возможно, но я не рекомендую использовать центральную платформу как серверную часть приложения.

Во-первых, потоковая передача имеет асинхронную природу. Для приложений, не имеющих требований к задержке, это не проблема, но при этом нужно будет убрать ограничение времени хранения, иначе можно потерять данные приложения. Во-вторых, создается тесная связь с приложением и целью распределения корпоративных данных. Это две разные задачи и они не должны смешиваться.

Хотя платформу потоковой передачи можно использовать как серверную часть для приложений, я рекомендую определить четкие правила относительно того, как поступать в таких ситуациях. Например, можно точно определить, какие области охватывает этот шаблон. Другой вариант — создать дополнительные платформы для определенных областей. Самый большой риск использования потоковой платформы в качестве серверной части — это создание интеграционной БД (<https://oreil.ly/SHDrh>).

Такой же принцип можно применить к потокам, получаемым из других потоков. Разрешение их использования другими потребителями может привести к образованию хаотической сети из тем, направленных в разные стороны. Такие спагетти-сети часто возникает из-за отсутствия контроля и политики, определяющих правила использования платформы. Хороший принцип — заставить потребителей сначала передать события в свои приложения, а потом передавать их обратно в потоковую платформу.

Потоковая передача как операционный конвейер

В главе 1 я рассказал о тенденциях к более тесному взаимодействию транзакционных и аналитических систем. Потоковая передача данных помогает решению этой задачи и служит основой для соединения систем, неспособных эффективно обрабатывать потоки из-за операционной деятельности, с современными аналитическими технологиями.

Дополнительная потоковая передача позволяет расширить операционную систему дополнительными компонентами. Вот несколько примеров.

- Расширение операционной системы дополнительной системой, хранящей исторические данные в памяти. Всякий раз, когда операционной системе нужен более подробный исторический контекст для принятия решения, платформа потоковой передачи объединяет операционную систему, систему исторических данных в памяти и аналитический механизм.
- Расширение операционной системы логикой принятия решений в реальном времени: когда клиенты покупают новые продукты, приложение может обратиться к финансовой системе и проверить наличие непогашенных долгов. Для этого используется потоковая передача.
- Использование источника команд для их сохранения и воспроизведения всего состояния системы. Пропущенные действия добавляются обратно в общий список и выполняются позже.



При построении операционных систем, поддерживающих асинхронный обмен большим количеством сообщений, вы, скорее всего, попадете в ситуацию, когда в периоды всплесков нагрузки системы будут получать данные быстрее, чем смогут обработать. Эта проблема называется противодавлением (<https://oreil.ly/8wbF5>), и, если ее не решить правильно, она может привести к исчерпанию ресурсов или даже к потере данных. Существуют разные стратегии решения этой проблемы. Добавление дополнительных ресурсов — одна из них. Другое решение — установить ограничения для очередей и отбрасывать или сохранять сообщения при превышении лимита в другом месте.

Та же архитектура может использоваться для разработки операционной системы, основывающейся на применении нескольких технологий. Потоковая передача здесь может поддерживать синхронизацию данных между различными БД или объединение и интеграцию данных. Архитектуры на основе микросервисов часто также полагаются на событийно-ориентированную архитектуру. Потоковая передача событий — это один из шаблонов, позволяющих микросервисам взаимодействовать друг с другом.

Гарантии и согласованность

Потоковая обработка сложнее пакетной. Если что-то пойдет не так при пакетной обработке, вы можете решить проблему, повторно доставив все данные и запустив пакетный процесс. УстраниТЬ сбои и несоответствия в потоковой передаче сложнее. В этом разделе мы рассмотрим некоторые аспекты, о которых следует помнить.

Уровень согласованности

Первый аспект — уровень согласованности, применяемый в потоковой архитектуре и приложениях. В потоковой передаче есть два типа согласованности.

- *Потенциальная согласованность.* События отправляются на платформу независимо от того, подтверждает их принимающая сторона или нет. Потенциальная согласованность — слабая гарантия, и может использоваться для некритических данных.
- *Строгая согласованность.* Строгая согласованность гарантирует, что вы не потеряете никакие события. Все они будут обработаны, а это значит, что некоторые из них могут быть доставлены несколько раз, пока гарантия не будет выполнена.

Любые проблемы, связанные с моделью строгой согласованности, вы можете решить с помощью нескольких архитектурных подходов. Можно использовать дополнительную БД для хранения всех данных и время от времени согласовывать для правильной синхронизации. Еще можно использовать дополнительные инструменты, вроде CDC, чтобы гарантировать правильную доставку всех изменений. В качестве альтернативы можно создать дополнительный сервис, обеспечивающий согласованность путем чтения, проверки и повторной отправки отсутствующих сообщений.

Поддержка строгой согласованности может быть трудной задачей. Если скорость событий настолько высока, что платформа потоковой передачи едва успевает за ними, может быть сложно или невозможно спроектировать систему так, чтобы все сообщения принимались правильно. В таких случаях вы можете убрать гарантии и согласиться с тем, что некоторые сообщения будут потеряны.

Семантики обработки «хотя бы один раз», «точно один раз» и «не больше одного раза»

Существуют разные семантики обработки в рамках модели потенциальной и строгой согласованности. Семантика «хотя бы один раз» гарантирует, что сообщения не будут потеряны, но не гарантирует их уникальность и доставку только один раз. Эта семантика допускается многократную обработку дубликатов событий.

Семантика «точно один раз», которая также называется моделью *доставки точно один раз*, гарантирует, что каждое событие будет доставлено один и только один раз. Сообщения не теряются, и дубликатов не существует. Для реализации этой семантики нужны дополнительные шаблоны повторной доставки. Некоторые инженеры используют дополнительные идентификаторы для проверки уникальности каждого сообщения.

Последняя семантика называется «*не более одного раза*» или *обработкой без гарантии*. Эта семантика не дает никаких гарантий относительно доставки сообщений. Они могут теряться или доставляться несколько раз.

Эти семантики могут применяться к обеим моделям, и потенциальной, и строгой согласованности. Например, модель потенциальной согласованности может использоваться для первоначальной доставки сообщений без подтверждения получения. Однако время от времени все сообщения сравниваются, очищаются и повторно доставляются, чтобы гарантировать семантику доставки точно один раз.

Порядок сообщений

Еще одна типичная проблема потоковой передачи — порядок сообщений. В некоторых случаях может потребоваться, чтобы все сообщения передавались в строго определенном порядке, например, в порядке создания. Kafka поддерживает такую возможность, но только для одного раздела. Если вам нужно больше разделов для масштабируемости, то порядок должен быть основан на дополнительном свойстве в сообщении, таком как `user_id` или `order_id`. Это может потребовать, чтобы доставляющая сторона внесла некоторые изменения в доставку. Другой подход — использовать все сообщения и переупорядочивать их на стороне потребителя.

Очередь недоставленных сообщений

На некоторых платформах потоковой передачи существует *очередь недоставленных сообщений* — дополнительная очередь для сообщений, , доставка которых завершилась неудачей. Сюда могут входить сообщения размером больше заданного предела; сообщения в неверном формате; сообщения, отправленные в несуществующую тему, или сообщения, которые не были извлечены в течение

заданного времени. В этих случаях сообщения не удаляются из системы, а помещаются в очередь недоставленных сообщений. В большинстве случаев разработчикам необходимо вручную проверить, что пошло не так. Одна из типичных причин — недостаточно хорошо протестированные изменения в приложении-поставщике. В таких сценариях сообщения необходимо переместить или снова отправить в исходную очередь¹.

Потоковое взаимодействие

Разработка интерфейсов между издателями и подписчиками требует принятия определенных технологических и проектных решений. Некоторые платформы диктуют, как приложения-поставщики должны подключаться к платформе и какой протокол они должны использовать. Другие платформы более открыты и поддерживают более широкий спектр протоколов, чтобы обеспечить прозрачное взаимодействие между несколькими языками программирования. Нейтральный способ и популярный выбор — использовать протокол, который кодирует сообщения с применением языка описания интерфейса (<https://oreil.ly/2CLeH>), для проверки формата данных и типов. Еще один важный аспект — выбор фреймворка, сериализации.

Фреймворки сериализации, как показано на рис. 5.16, играют важную роль в совместимости, управлении версиями и переносимости при перемещении данных между интерфейсами. С этой точки зрения такие фреймворки заботятся о последовательном преобразовании данных в стандартизованный формат. После преобразования многие системы, языки программирования и другие фреймворки могут начать использовать данные. Эти фреймворки сериализации также дополняют обработку данных такими функциями, как сжатие и безопасность (шифрования). Это делает сериализацию идеальной для передачи данных в хранилище только для чтения. В числе популярных фреймворков сериализации и протоколов можно назвать Apache Avro, Protocol Buffers, Apache Thrift, MQTT и AMQP.

Еще один способ создания и использования сообщений с потоковых платформ — использование REST API с поддержкой JSON- и XML-сообщений. Это хороший выбор для онлайн-каналов или веб-окружений. Kafka не поддерживает эту форму связи без установленного REST-прокси (<https://oreil.ly/vkjNc>). Преимущества JSON — гибкость и отсутствие схемы, но этот формат имеет один серьезный недостаток: он не обеспечивает совместимости из-за отсутствия обязательной схемы кодирования.

¹ Xia N. Building Reliable Reprocessing and Dead Letter Queues with Apache Kafka. Uber, 16 февраля 2018 г. <https://eng.uber.com/reliable-reprocessing>.

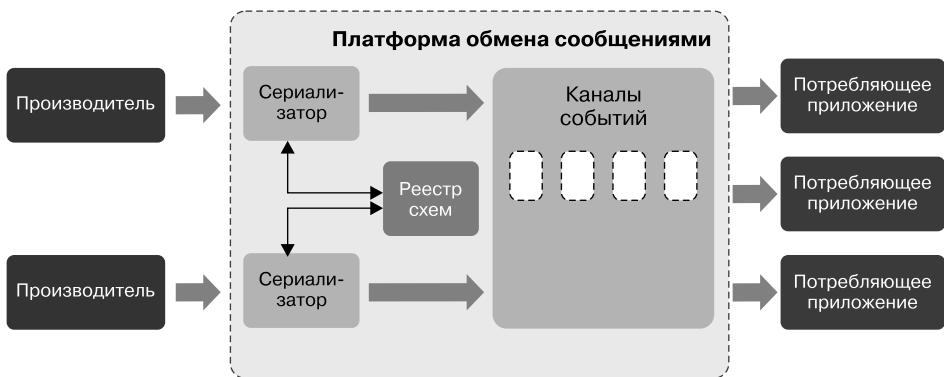


Рис. 5.16. Когда производители публикуют информацию, сериализатор регистрирует и использует реестр схемы, чтобы проверить, была ли схема уже зарегистрирована и используются ли правильная версия и схема (совместимость). Обычно каждая схема хранится под уникальным именем, а каждая версия определяется уникальным идентификатором

Метаданные для моделей управления и самообслуживания

Архитектура потоковой передачи, кроме поддержки потоков, включает дополнительные возможности и требования к метаданным. Наиболее важным из них является реестр тем (или реестр схем), который используется по ряду причин. Вот некоторые из них.

- *Очередь сообщений и регистрация владельца темы.* Данные всегда кому-то принадлежат. Очередь сообщений и реестр тем должны хранить информацию обо всех владельцах событий. Этот реестр также можно использовать для классификации очередей сообщений и тем как внутренних или общедоступных.
- *Схема управления документами.* Очередь сообщений и реестр тем должны задокументировать структуру схемы сообщений. Для XML и JSON это логические схемы XML или JSON. Форматы сериализации, такие как Avro и Protobuf, используют собственный язык определения интерфейса (interface definition language, IDL).
- *Управление версиями.* Схемы должны иметь версии. В противном случае критические изменения могут напрямую повлиять на пользователей событий. Ожидается, что управление версиями для схем будет работать так же, как для RDS и API.

- *Происхождение.* Ожидается, что события будут поддерживать историю происхождения, что потребует включения во все события уникального идентификационного номера происхождения. Этот номер может соответствовать, например, названию темы или быть уникальным для каждого события. Ожидается, что потребители, создающие новые события на основе полученных, будут включать в них старый идентификационный номер происхождения, чтобы всегда можно было выяснить их происхождение.

Метаданные потоковой платформы важны для улучшения целевой операционной модели с помощью самообслуживания. Например, вы можете разрешить областям сбрасывать смещения, очереди самообслуживания и темы и автоматически утверждать их. Или вам может понадобиться, чтобы области создавали свои шаблоны REST Proxy и автоматически развертывали их с использованием правильных политик безопасности. Вы также можете разрешить областям фильтровать данные перед передачей сообщений разным потребителям. Я рекомендую включать эти шаблоны один за другим, помня о самообслуживании.

Большинство таких шаблонов основаны на платформе с (центральным) реестром. Известные поставщики в этой области — Confluent Platform (<https://oreil.ly/S-mhX>) и Nakadi (<https://nakadi.io/>), основанные на Apache Kafka и Streamlio (<https://streamlio.io/>) (который, в свою очередь, основан на Apache Pulsar). Еще одна компания, которая создала отличный интерфейс для управления Kafka и микросервисами, безопасностью и квотами, — это Lenses (<https://lenses.io/>).

Итоги главы

Предприятия уже осознали силу событий. С их помощью мы можем соединять приложения, объединять данные, оцифровывать процессы и совершенствовать внутренние и внешние взаимодействия. Они позволяют нам понять, как системы реагируют друг на друга. Благодаря этим знаниям мы можем оптимизировать наши повседневные бизнес-операции.

Потоковая обработка сложна. Чтобы постоянно собирать данные и принимать потоки, в архитектуру нужно добавить много дополнительных компонентов. Учитывая сложность и разнообразие ландшафта, вы можете использовать множество различных инструментов, фреймворков и сервисов, например, для обеспечения единовременной обработки. Здесь важно, чтобы производители событий отвечали за отправку данных: их роль заключается в предоставлении стабильных и повторно используемых данных с использованием их собственного контекста.

В основе архитектуры потоковой передачи лежит платформа, позволяющая обогащать, преобразовывать, интегрировать и анализировать данные. В за-

висимости от выбранной вами платформы вы можете получить широкие возможности работы с окнами и API для взаимодействия с потоками. На уровне предприятия вы должны четко отделять частные каналы связи и потоки между приложениями, чтобы ваши потоки не попали в тесно связанный распределенный монолит. Ключевым моментом здесь является реализация четкого управления: когда меняется контекст, меняется и право собственности. Также важно сосредоточиться на корпоративных ключевых контекстах, поскольку они обеспечивают согласованность для потребителей при объединении нескольких потоков. Мы рассмотрим эту тему более подробно в главе 9.

На стороне потребителя можно столкнуться с еще большими сложностями, потому что выбор вариантов использования и шаблонов огромен. В зависимости от потребностей приложения и предметной области вы можете использовать дополнительные фреймворки обработки потоковой передачи, аналитические возможности или дополнительные конвейеры для внутреннего распределения потоков. Кроме того, области также могут взаимодействовать между собой и обогащать свои потоковые сообщения с помощью API.

Возможности потоковой передачи и обработки событий для обмена сообщениями и событиями между поставщиками и потребителями должны быть помещены в потоковую архитектуру, где они играют важную роль в распределении сообщений между различными окружениями и поддержании RDS в актуальном состоянии. При подключении нескольких брокеров событий вы можете соединить вместе множество распределенных и независимых приложений на нескольких платформах и в облачных окружениях. Этот тип архитектуры также известен как *сетка событий*.

В следующей главе показано, как все многочисленные архитектуры работают вместе.

ГЛАВА 6

Соединение точек

Эта глава кратко рассматривает архитектуры с различных точек зрения, затем связывает их с такими дисциплинами управления данными, как регулирование, моделирование и управление метаданными. Они взаимосвязаны, поэтому важно гарантировать согласованность и единообразие обеспечения безопасности, регулирования, управления метаданными и моделирования данных во всех архитектурах.

После этого мы обсудим проектирование данных и интерфейсов для всех архитектур. Вы узнаете, когда и какой шаблон интеграции использовать, какие комбинации можно создавать и что лучше всего работает в гибридных и мультиоблачных моделях. Я расскажу о важных стандартах обнаружения и взаимодействия, о преимуществах задания принципов для устойчивых и повторно используемых данных, улучшающих общее потребление данных и устраниющих из областей повторяющуюся работу. Еще вы узнаете, почему такие огромные усилия тратятся на создание архитектуры, управляемой метаданными. Наконец, я покажу, как добиться семантической согласованности, подключив конечные точки области к уровню абстракции, который описывает каждый из уникальных наборов данных и элементов.

Кратко об архитектурах

Главу 2 мы начали с обзора целостной картины, включающей все коммуникационные потоки. Вы узнали, что внутренняя сложность области и приложения должна быть скрыта от других областей и что согласованные данные, оптимизированные для потребления, должны предоставляться через уровень данных.

Все приложения, которые должны взаимодействовать и обмениваться данными, собраны вместе на уровне данных (рис. 6.1) и остаются разделенными. После того как область выполняет свою работу и данные подготавливаются

к использованию через одну из архитектур интеграции, эти данные становятся доступными для всех.

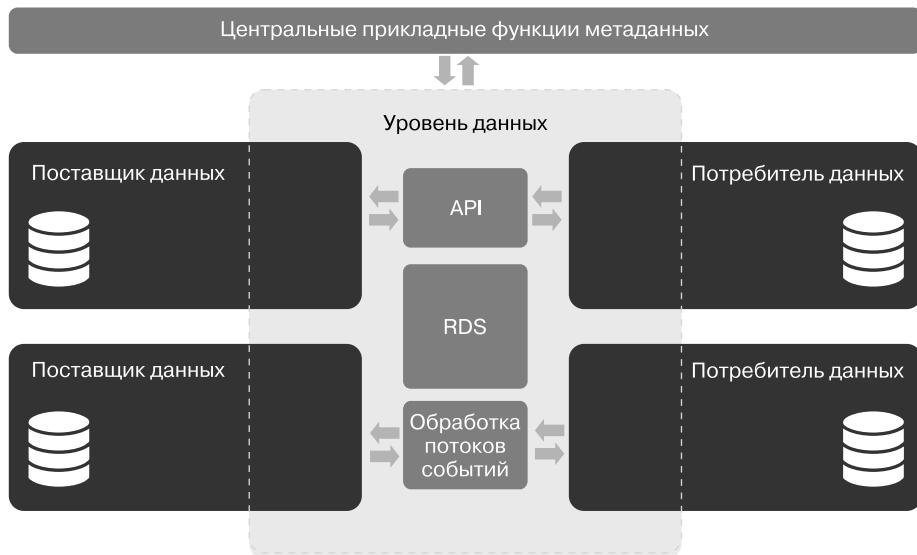


Рис. 6.1. Напоминание об общем представлении, которое иллюстрирует, как объединяются три разные архитектуры и метаданные

Мы подробно рассматривали, как каждая часть архитектуры интеграции способствует обмену данными. Эти три архитектуры составляют общий набор корпоративных инструментов. Но они также поддерживают подход, соответствующий назначению, и предлагают различные шаблоны на выбор.

Архитектура RDS

Архитектура RDS, как вы узнали из главы 3, в основном обеспечивает варианты использования, связанные с большими объемами данных, такие как бизнес-исследования и аналитика. Она играет важную роль в управлении качеством данных и их жизненным циклом. Архитектура RDS предназначена для активного чтения и надежного хранения данных. Хранилища данных только для чтения — одна из важных частей этой архитектуры, которая может включать API, не требующие строгой согласованности, для поддержки работы приложений в реальном времени. Архитектура, как вы поняли, имеет много общего с CQRS. Чтение обслуживается из специальной БД, доступной только для чтения (read-only data store, RDS), и не обслуживается с помощью базы данных приложения — обычно операционной или транзакционной системы.

Архитектура API

Архитектура API (как вы узнали из главы 4) ориентирована на синхронную связь в реальном времени и может использоваться для связи между устаревшими системами и современными приложениями. Она может предоставлять как данные, так и бизнес-функции. Еще эта архитектура использует модель «запрос/ответ», типичную для клиент-серверных архитектур. Для простоты этот шаблон часто реализуется синхронно. Хотя большинство операций чтения должны выполняться из RDS, из этого правила есть исключение. RDS по своей природе асинхронны, поэтому для согласованных — очень точных — данных этот шаблон не работает и необходимо использовать архитектуру API.

Архитектура потоковой передачи

Архитектура потоковой передачи (обсуждаемая в главе 5) обеспечивает событийно-ориентированные взаимодействия и позволяет преобразовывать данные в реальном времени и уведомлять другие приложения. Потоковая передача событий — это не диалог, а односторонний канал связи, который отправляет данные слушателям. Вместо того чтобы спрашивать: «Какие данные я могу запросить или извлечь из приложения?» — используется модель: «Какие проходящие мимо данные могут мне пригодиться?» Благодаря потоковой передаче можно быстро реагировать на данные. Наконец, с ее помощью можно превратить платформу в базу данных потоковой передачи событий, которая одновременно действует как RDS и поддерживает модель «запрос/ответ».

Архитектура API и потоковая архитектура поддерживают разные рабочие процессы для разных сценариев использования. Базовые архитектуры различны и созданы для разных целей. Управляемые событиями архитектуры также можно использовать для распределения событий, которые передают состояние приложений, чтобы включить репликацию данных, например, при построении RDS. Этот шаблон также известен как *передача состояния с переносом событий*.

Настоящий потенциал кроется в том, что все команды могут независимо развивать свои предметные области и выбирать наиболее подходящие шаблоны интеграции приложений. Разрозненные хранилища не будут создаваться, потому что приложения и данные управляются в пределах границ, основанных на согласованности бизнес-потребностей (бизнес-возможностей). Двухточечных интерфейсов обычно избегают, потому что все каналы связи заключены в архитектуру с четкими принципами проектирования. Каждая область экспортит приложения только один раз. Эти ключевые отличия делают архитектуру хорошо масштабируемой.

Усиливающие шаблоны

Эти архитектуры не замкнуты на самих себе — их можно комбинировать для усиления друг друга. В этом разделе я покажу ряд архитектурных шаблонов и расскажу, какой синергетический эффект можно получить, комбинируя разные архитектуры.

- *Маршрутизация в шлюзах и таблицы индексов.* RDS и API можно объединить для направления трафика запросов в RDS и команд в операционные системы. Этот шаблон похож на CQRS (подробно описанный в разделе «Использование RDS для чтения в реальном времени и активного чтения» на с. 149). Для поддержки использования в режиме реального времени и эксплуатации можно объединить шаблон маршрутизации в шлюзах (<https://oreil.ly/q9NFI>) с таблицами индексов (https://oreil.ly/_5F_H) — индексами полей в хранилищах данных, на которые часто ссылаются запросы. Эти шаблоны могут повысить производительность запросов, позволяя приложениям быстрее находить данные и извлекать их из хранилища только для чтения.
- *Распределенные RDS.* Вы можете хранить наиболее часто запрашиваемые данные ближе к потребителям, копируя состояние приложения в RDS в режиме реального времени. Этот шаблон похож на создание распределенного кэша или материализованных представлений. Он помогает смягчить действие тяжелых нагрузок чтения на операционную систему и уменьшить объем трафика при переносе рабочих нагрузок в облако. В этой модели кэши могут структурировать данные любым способом для поддержки различных шаблонов доступа каждого приложения. Этот подход (рис. 6.2) также можно адаптировать к шаблону маршрутизации в шлюзах API с целью разделения и направления строго согласованных запросов, команд и возможных операций чтения.
- *Выравнивание нагрузки на основе очередей.* Переполнение API слишком большим количеством одновременных запросов может привести к сбоям в работе сервисов. Из-за перегрузки сервисы могут несвоевременно отвечать на запросы. Чтобы решить эту проблему, можно объединить архитектуру API и потоковую архитектуру и создать очередь, действующую как буфер для запросов к архитектуре API. Этот шаблон также известен как *выравнивание нагрузки на основе очередей* (<https://oreil.ly/y2IjE>).
- *Уведомитель.* В главе 3 я упоминал, что службы уведомления для RDS могут понадобиться для информирования потребителей о готовности данных к использованию. Этот сервис позволяет автоматизировать конвейеры. Уведомления, как вы уже наверняка догадались, должны использовать потоковую архитектуру.

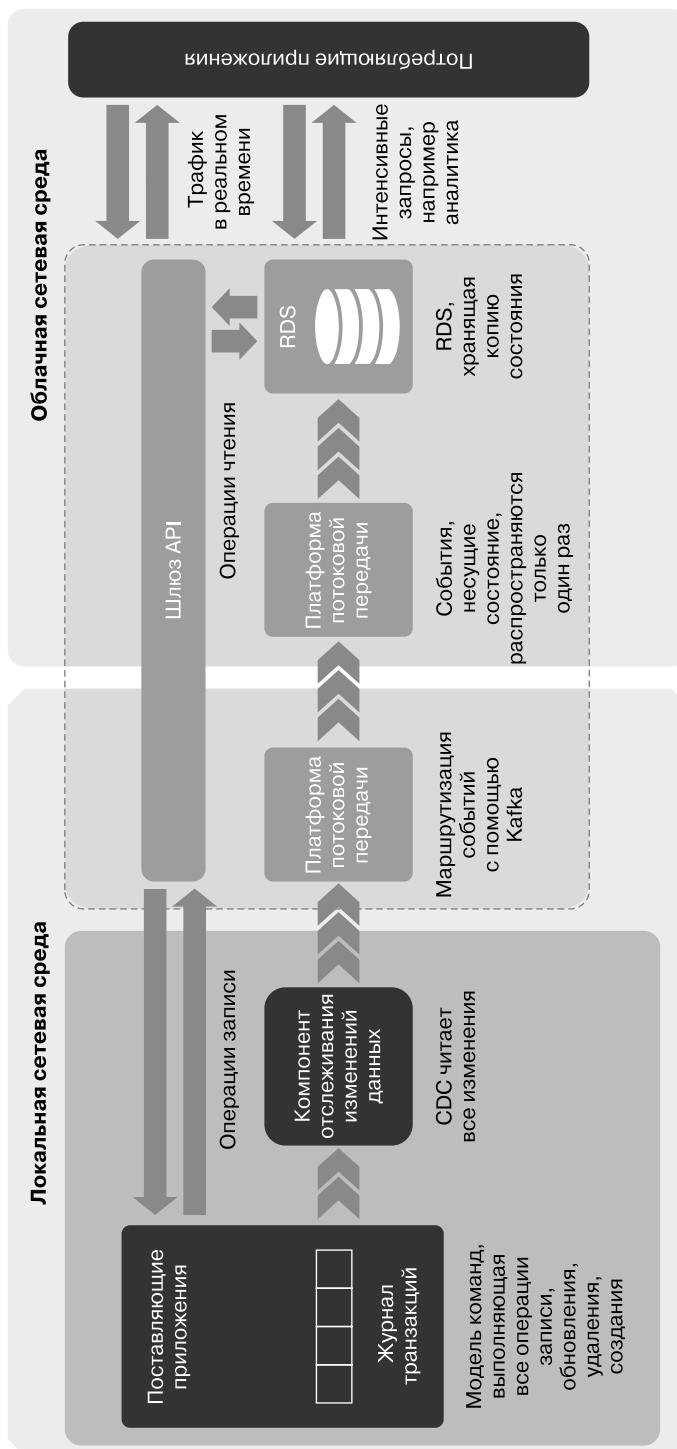


Рис. 6.2. RDS, платформы потоковой передачи и шлюзы API могут быть объединены для уменьшения объема трафика облачка

Еще один интересный шаблон — это преобразование медленно меняющихся измерений, которые обнаруживают и обрабатывают изменившиеся строки. Вы можете расширить этот метод, охватив не только обработку исторических данных и запись в новый файл или БД, но и создание событий. Эти события содержат исторические и текущие значения атрибутов, что позволяет потребителям реагировать на изменения вместо обработки всех данных.

Прием потоковой передачи (обсуждается в главах 3 и 5) — это подход с использованием платформы потоковой передачи для передачи состояния приложения и заполнения RDS. Этот шаблон полезен при создании распределенных RDS, охватывающих несколько сетевых местоположений.

Архитектурные шаблоны в этой главе обычно используются на уровне данных. Есть много других архитектурных шаблонов, применяемых на стороне поставщика или потребителя. Чтобы ознакомиться с подробным списком шаблонов, которые можно использовать в архитектуре, обратитесь в центр архитектуры Microsoft Azure (<https://oreil.ly/S-9NU>).

Стандарты корпоративной совместимости

В случае децентрализации управления владением и распределением данных важно стандартизировать согласованность взаимодействий между областями на уровне предприятия. В этом разделе обсуждается, как это сделать, включая обработку изменений интерфейса, управление зависимостями между областями, обеспечение доступности и адресации данных, а также управление распределением данных по сетевым окружениям.

Важно установить четкие принципы взаимодействия и разработки для управления данными. RDS, API и потоковая передача событий не сильно отличаются друг от друга. Все эти конечные точки могут использоваться для передачи и распространения одних и тех же предметных данных, поэтому все они должны следовать одним правилам.

Конечные точки устойчивых данных

Приложения, которые обращаются к данным друг друга напрямую, всегда страдают проблемой связанности¹. Под *связанностью* подразумевается высокая степень взаимной созависимости. Например, любое изменение структуры данных

¹ Под связанностью в контексте программной инженерии понимается степень связи между программными компонентами. (На сайте connascence.io вы найдете справочник по различным типам связывания.) Компоненты программного обеспечения считаются связанными, если изменение одного требует изменения другого (других) для поддержания общей правильности системы.

или протокола окажет прямое влияние на другие приложения. В случаях, когда многие приложения тесно связаны друг с другом, иногда может наблюдаться каскадный эффект. Даже небольшое изменение в одном приложении может привести к изменениям во многих одновременно. Вот почему некоторые архитекторы и инженеры-программисты избегают создания связанных архитектур.

В масштабируемой архитектуре основная связь между приложениями от поставщиков и потребителей данных находится на уровне данных¹. Мы стремимся к тому, что обычно называют *сильным внутренним сцеплением и слабой внешней связанностью*. В идеальной ситуации поставщик и потребитель данных полностью автономны и не знают друг о друге. Чтобы достичь такой «разделенности», нужно установить ряд принципов связывания. Давайте разберемся, что это значит.

СВЯЗАННОСТЬ В МАСШТАБИРУЕМОЙ АРХИТЕКТУРЕ

Да, в масштабируемой архитектуре есть связанность! При создании интерфейсов всегда появляются взаимосвязи. Чем больше потребителей, тем больше связей. В хранилищах данных обычно образуется связь между всеми сторонами, потому что данные всегда согласовываются и интегрируются перед потреблением. В масштабируемой архитектуре связь находится на уровне данных. В отличие от хранилища только поставщик связан с потребителями, которые потребляют данные от поставщика.

В масштабируемой архитектуре важна совместимость схем областей при использовании интерфейсов друг друга. Как известно, структура БД со временем может меняться. По мере устаревания создаются новые столбцы, а старые удаляются. Столбцы можно преобразовать в несколько столбцов, например разбить строку с адресом на отдельные столбцы (улица, номер дома, город и т. д.). К сожалению, такое изменение схемы может нарушить работу интерфейса. Если, например, имя столбца изменяется, запросы к этому конкретному столбцу будут завершаться ошибкой.

Эволюция схемы, процесс внесения изменений в структуры БД и поддержание совместимости схем — важный аспект управления данными. Без него архитектура начнет разрушаться. В идеале области будут беспрепятственно обрабатывать данные со старой и новой схемами. В реальности все, конечно, сложнее. Эволюция схемы предполагает несколько видов совместимости: то есть способность одного компьютера, программного обеспечения или другого компонента работать с другим.

¹ Уровень данных действует как один большой антикоррозийный слой (<https://oreil.ly/VoZiu>): слой фасада или адаптера между различными подсистемами или приложениями, не имеющими одинаковой семантики. Этот уровень преобразует обмен данными между приложениями, позволяя одному приложению оставаться неизменным, в то время как другое избегает ущерба для своего проекта и технологического подхода.

- *Обратная совместимость.* Обратная совместимость, как показано на рис. 6.3, означает, что области, использующие новую схему, могут читать данные, созданные с помощью предыдущей схемы. Примером обратно совместимого изменения является удаление поля и установка для него значения по умолчанию. Это позволяет области продолжать выполнять SQL-запросы (например, с использованием SQL Server или Apache Hive). Если для столбца «цвет» будет установлено значение по умолчанию «оранжевый», то области все равно смогут запрашивать эти данные после удаления поля — они будут получать значение по умолчанию «оранжевый».

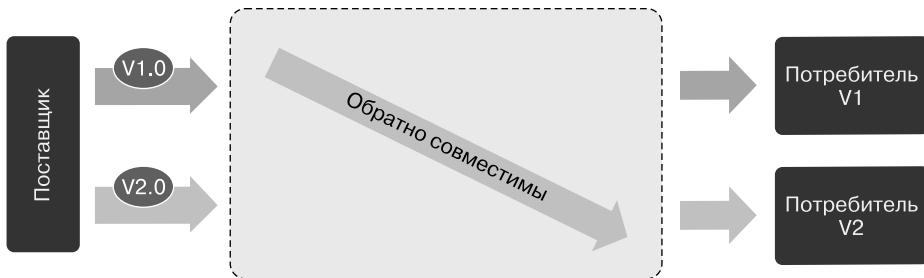


Рис. 6.3. Интерфейс обратно совместим, если приложение-потребитель, использующее более новую схему (версия 2), может читать данные, созданные с использованием более старой схемы (версия 1)

- *Прямая совместимость.* Прямая совместимость, как показано на рис. 6.4, означает, что данные предоставляются с новой схемой, но могут использоваться областями с предыдущей (старой) схемой. Пример изменения схемы, не нарушающего прямой совместимости, — добавление новых или удаление необязательных полей. Добавление дополнительных полей не повлияет напрямую на области, потому что нет отсутствующих данных.



Рис. 6.4. Интерфейс является прямо совместимым, если пользовательское приложение, применяющее старую схему (версия 1), может читать данные, созданные с использованием более новой схемы (версия 2)

- *Полная совместимость.* Полная совместимость означает, что схемы имеют как обратную, так и прямую совместимость. Схемы развиваются так, что старые данные можно читать с помощью новой схемы, а новые данные — с помощью старой схемы.

Конечные точки интерфейса представляют систему-источник области. Это означает, что система-источник может изменяться независимо, без изменения интерфейсов. Такое разделение позволяет областям развивать схемы интерфейсов без координации с другими областями. Разработчики могут, например, провести рефакторинг операционной системы без непосредственного изменения RDS. Такое возможно, пока структура интерфейса остается прежней.

УПРАВЛЕНИЕ ВЕРСИЯМИ ДАННЫХ И ИНТЕРФЕЙСОВ

Часто совместимость обрабатывается с помощью *управления версиями*. При каждом незначительном изменении номер версии увеличивается не намного, тогда как при значительных изменениях, влияющих на совместимость, номер версии обычно увеличивается на большое число. Лучше всего использовать семантический подход к управлению версиями (<https://semver.org/>) *В.КРУПНОЕ_ИЗМЕНЕНИЕ.НЕБОЛЬШОЕ_ИЗМЕНЕНИЕ.ИСПРАВЛЕНИЕ*.

Протокол тоже может изменяться: например, структура метаданных для описания схемы или новая информация в метаданных, необходимая для конфиденциальности и безопасности. Я рекомендую использовать управление версиями на уровне протокола, что может привести к тому, что в интерфейсе будут отражены две версии типа: для самого интерфейса и для используемого протокола.

Если нужно значительно изменить систему и интерфейс с возможной потерей совместимости, то следует создать новый интерфейс. Старый должен продолжать поддерживаться некоторое время, чтобы дать другим областям возможность для миграции. Один из способов управления и предотвращения нарушения интерфейсов после изменений — использование *контрактов на доставку данных*, также известных как *контракты на обслуживание*. Эти контракты очень важны для стабильной архитектуры.

Контракты на доставку данных

Контракты на доставку данных — один из самых важных аспектов масштабируемой архитектуры. Как только конечные точки данных области, такие как RDS, станут популярными и будут широко использоваться, быстро возникнет потребность в управлении версиями, совместимостью и развертыванием. Без этих правил вероятность повторного использования может быть низкой, а интерфейсы могут сломаться.

На самом базовом уровне я рекомендую документировать контракты для всех интерфейсов, включая схемы (форматы сообщений и данные) и типы передачи, а также их связь с приложениями. Чтобы контракты можно было быстро находить и повторно использовать, они должны храниться в *репозитории метаданных*.

На более продвинутом уровне следует сделать данные контрактов общедоступными для всех областей через интерфейсы (например, API) и информационные панели. Это позволит областям автоматизировать процедуры тестирования в конвейерах непрерывной интеграции и развертывания. Зная, какие конкретно части интерфейсов используются (например, столбцы и объекты), команды разработчиков смогут тестировать подпрограммы для проверки любых данных, которые будут доставлены. Проверка соответствия структуры (имен столбцов и типов значений) гарантирует совместимость.

СЛУЖБА КВИТИРОВАНИЯ В RDS

Чтобы улучшить модель проверки RDS в архитектуре, вы можете реализовать службу квитирования. Она возвращает код подтверждения, проверки или ошибки после доставки данных. Все операции доставки данных в RDS также должны регистрироваться и контролироваться. Этот подход показан на рис. 6.5, где доставка данных проверяется на соответствие контрактам в репозитории метаданных. Поля или таблицы, которые подлежат потреблению, не могут нарушать совместимость.

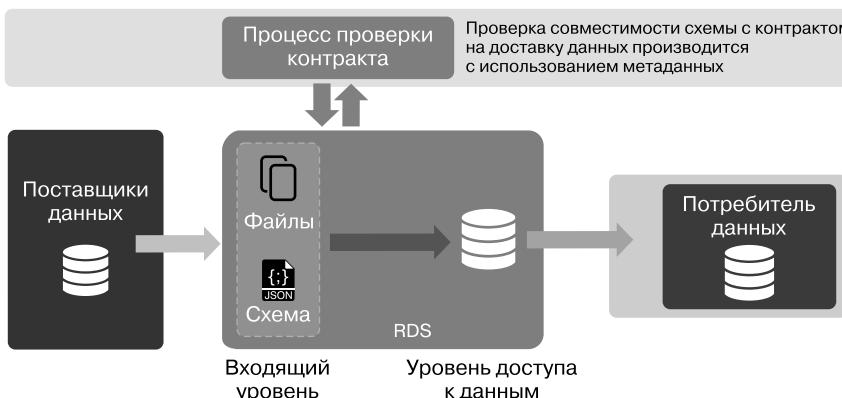


Рис. 6.5. При проверке данных на совместимость необходимо использовать контракты на поставку данных

Понимание, которое дает модель масштабируемой архитектуры в отношении совместимости, управления версиями, возможности обнаружения и повторного использования для каждого интерфейса, очень поможет общей гибкости организации. В следующем разделе мы рассмотрим принцип обеспечения доступности и адресации данных области.

Доступные и адресуемые данные

Для доступа к данным нужно иметь информацию о местоположениях ресурсов, хранилищ, событий и API, включая протоколы и форматы данных. Чтобы еще больше упростить доступ к данным, можно опубликовать все унифицированные идентификаторы ресурсов (Uniform Resource Identifiers, URI) в центральном репозитории или системе управления конфигурацией, такой как Consul (<https://www.consul.io/>)¹. Доступность URI позволяет предметным областям автоматизировать доступ к данным с помощью настроек и сценариев.

Для дополнительной развязки вы можете пользоваться системой доменных имен (Domain Name System, DNS) (<https://oreil.ly/xutxs>), чтобы изменить IP-адрес и физическое местоположение базового приложения, не нарушая интерфейсы и не задерживая доступ к данным областей.

Принципы пересечения сетей

Большая часть всего обмена данными обычно происходит в пределах предприятия, близко к большинству приложений. *Теория гравитации данных* (<https://oreil.ly/Abz9I>) Дэйва Маккрори (Dave McCrory) рассматривает данные как сущности, имеющие массу и, как и любой физический объект, гравитацию, которая притягивает сервисы и приложения ближе к данным для преодоления проблем с пропускной способностью и/или уменьшения задержки доступа.

Хотя теория гравитации данных верна, все начинает меняться при перемещении приложений из локальной среды в публичное облако, использовании SaaS, потреблении открытых данных и взаимодействии с внешними компаниями. Централизованное хранение всех данных приводит к большой проблеме: если вы постоянно копируете одни и те же данные, пропускная способность сети быстро станет вашим самым страшным врагом.

Отправляйте копии данных к месту потребления

Решение этой проблемы — распределить обработку с отслеживанием состояния там, где происходит потребление. Архитектура потоковой передачи, хранилища состояний (см. врезку «Хранилища состояний» на с. 171) и синхронизация RDS — это шаблоны репликации данных в децентрализованной и распределенной среде. После репликации данные можно снова распределить в пределах нового сетевого окружения, не копируя их снова и снова. Чтобы разобраться

¹ URI — это уникальная строка символов, используемая для идентификации конкретного ресурса. Аналогичный способ идентификации используется во Всемирной паутине. Например, URL-адреса идентифицируют ресурсы, представленные в форме HTML.

в этом подходе, давайте рассмотрим другую эталонную архитектуру, которая использует сети в качестве точки обзора.

На рис. 6.6 вы заметите несколько важных моментов: платформы потоковой передачи и хранилища данных только для чтения используются дважды для потребления данных. Тогда как шлюз API — только один раз с каждой стороны. Причина в том, что потоковые платформы и RDS могут хранить состояние приложения, а архитектура API в основном использует шаблон «запрос/ответ» — синхронную форму связи без сохранения состояния. Вызовы сервисов через HTTP поддерживают соединение открытым и ждут, пока не будет доставлен ответ; затем данные исчезают. Чтобы избежать проблем с задержкой и пропускной способностью, я рекомендую развернуть уровень кэширования для шлюзов API на другой стороне. Уровень кэширования — это компонент, который хранит результат API, обычно в памяти или в высокопроизводительной БД, благодаря чему последующие похожие запросы могут обслуживаться намного быстрее и в том же месте.

Еще один вывод, который можно сделать, глядя на рис. 6.6, в том, что разделение всегда происходит близко к месту происхождения или создания данных. Это сделано намеренно: иначе трафик будет пересыпаться по сети вперед-назад. Например, если приложения, которые предоставляют API, находятся в локальной системе, а шлюз API — в облаке, то все обращения локальных приложений к API сначала будут передаваться в шлюз API в облаке, затем обратно в локальную сеть, потом снова в облако и, наконец, обратно в локальную сеть. Такие петли маршрутизации крайне неэффективны.

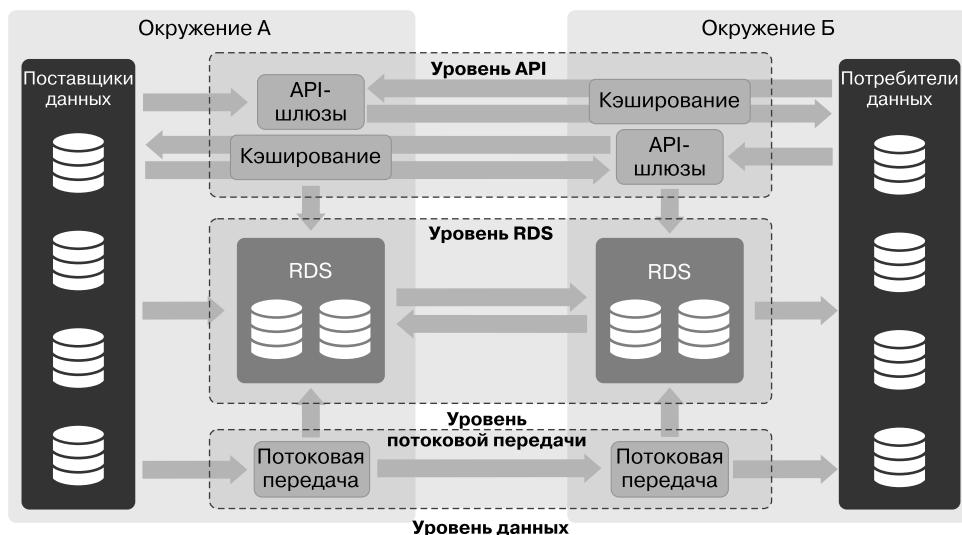


Рис. 6.6. Архитектуры охватывают сетевые окружения. Разделение должно происходить как можно ближе к данным

Думайте в духе асинхронности

При пересечении сетей возникает еще один риск: они могут выйти из строя, и в этом случае приложения не смогут получить доступ к их данным. Чтобы изолировать эту проблему и сделать архитектуру более устойчивой, нужно начать мыслить в духе *асинхронности*. Это подойдет не для каждого сценария, но при отсутствии жестких требований к задержкам асинхронная связь имеет ряд преимуществ.

- Проблемы с сетью, такие как низкая производительность и большая задержка, не влияют или оказывают минимальное влияние на асинхронную связь.
- Асинхронную связь легче масштабировать, если вдруг нагрузка увеличится.
- Асинхронная связь делает взаимодействие приложений более устойчивым к временным сбоям сети.
- С помощью асинхронной связи можно реализовать важные функции для повышения устойчивости — механизмы повторных попыток, резервные сценарии и прерыватели цепи, которые останавливают каскадные эффекты.

Асинхронная связь имеет также ряд недостатков. Поскольку взаимодействия происходят не в реальном времени, их труднее отлаживать и исследовать, если что-то пойдет не так. Когда вызов запрос/ответ терпит неудачу, это обычно сразу заметно. Если асинхронная система не работает в течение длительного времени, это может привести к появлению новых проблем. Для асинхронной связи также может потребоваться добавить в приложение дополнительные компоненты. Например, может понадобиться защитить асинхронные потоки сообщений и постараться избежать риска их потери с помощью очередей недоставленных сообщений или автоматических прерывателей цепи¹.

Разделение предметных областей

Синхронное и асинхронное разделение тесно связано с предметно-ориентированным проектированием (см. подраздел «Предметно-ориентированное проектирование» на с. 46), которое можно использовать для проведения логических границ в ландшафте корпоративных приложений. Ограниченные контексты используются для задания границ областей с большим внутренним сцеплением. В этом разделе я сформулирую некоторые дополнительные принципы для моделей синхронного и асинхронного взаимодействия. В качестве конкретного примера воспользуемся другой эталонной моделью. На рис. 6.7 изображено пять областей. Каждая фокусируется на определенной сфере бизнеса. Области А и Б расположены в физической среде 1, В и Г — в физической среде 2, а Д охватывает две физические среды.

¹ Автоматический прерыватель цепи (<https://oreilly/XFYvP>) — шаблон проектирования, используемый для обнаружения отказов и заключающий в себе логику предотвращения его постоянного повторения.

При анализе совместимости данных выделяются следующие закономерности.

1. Все взаимодействия между приложениями осуществляются в пределах области. В этом случае контекст не меняется, но из-за большого внутреннего сцепления эти приложения должны развертываться вместе. Например, они должны использовать одну и ту же группу ресурсов или виртуальную частную сеть.
2. Взаимодействия протекают в физической среде, но пересекают границы областей. При пересечении границ контексты меняются. Хотя все взаимодействия между приложениями происходят в одной физической среде, компьютерные ресурсы каждой области должны развертываться в разных группах ресурсов или виртуальных частных сетях.
3. Взаимодействия осуществляются через границы физических сред и областей, что увеличивает вероятность возникновения ошибок в приложении. Необходимо предпринимать дополнительные меры по увеличению отказоустойчивости.
4. Взаимодействия осуществляются в пределах области, но данные копируются между сетями. Ожидается, что контекст приложения останется прежним, но существует зависимость от сети, поэтому необходимо учитывать дополнительные сетевые аспекты.
5. Взаимодействия осуществляются с (неконтролируемыми и неуправляемыми) внешними сторонами, такими как поставщики SaaS и внешние потребители данных. Данные покидают корпоративную среду, поэтому важно предпринимать дополнительные меры по увеличению безопасности сети.

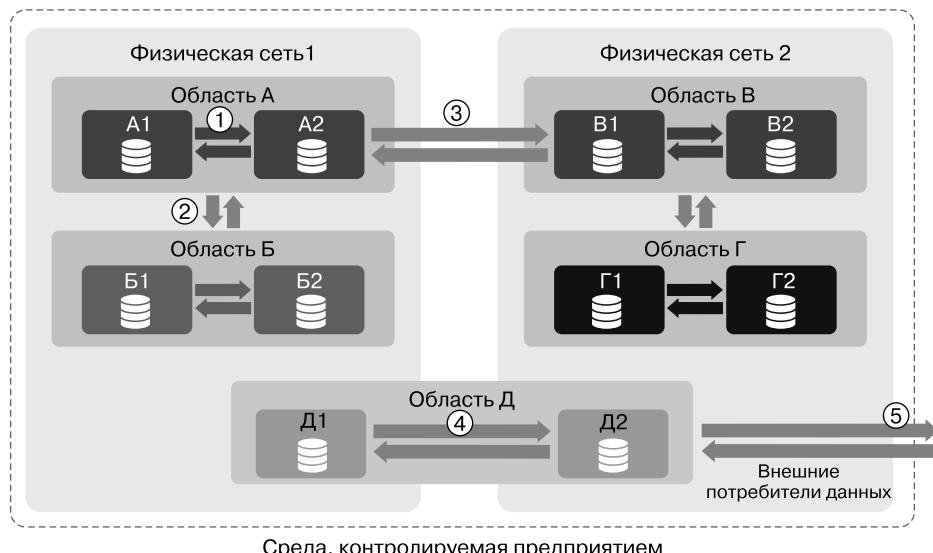


Рис. 6.7. Различные способы разделения областей

На основании этих закономерностей можно сделать некоторые выводы. Я перечислил самые важные в табл. 6.1.

Таблица 6.1. Соображения по поводу различных шаблонов областей

	Интеграция	Способ взаимодействия	Безопасность
1. Связь между приложениями	Задача интеграции должна решаться общей	Синхронный и асинхронный, с одинаковым эффектом	Согласно внутренней политике
2. Связь между областями	Необходимо использовать архитектуры интеграции	Синхронный и асинхронный, с одинаковым эффектом	Согласно политике предприятия
3. Связь между областями через границы сетей	Необходимо использовать архитектуры интеграции	Желательно асинхронный	Согласно политике предприятия
4. Связь между приложениями, но через границы сети	Задача интеграции должна решаться общей	Желательно асинхронный	Согласно внутренней политике
5. Связь между внутренними приложениями и внешними потребителями	Необходимо использовать архитектуры интеграции	Синхронный и асинхронный, с одинаковым эффектом	Усиленная и строгая

Чтобы не было проблем с подключением к сети и платформе, я рекомендую по возможности использовать асинхронную связь. Другой принцип в том, что в ограниченных контекстах всегда должен использоваться уровень данных с его набором принципов. Дополнительно необходимо сохранять все метаданные. Для связи с внешними потребителями нужны дополнительные меры безопасности, такие как защита от DDoS-атак. Для микросервисов (см. раздел «Микросервисы» на с. 136) я рекомендую следовать тем же правилам. Хотя они обычно развертываются в очень больших инфраструктурах, может возникнуть соблазн напрямую общаться с ними из другой области или ограниченного контекста. Области должны знать, что понимание изменений недоступно, потому что обычно иная группа или область заботится о другом наборе микросервисов. Без дополнительной связки повышается риск нарушений в работе интерфейсов.

Исключения

Бывают ли ситуации, когда не следует использовать уровень данных при распределении данных по областям или приложениям? Да, но только одна. Единственным исключением являются приложения, критичные к задержке, для которых задержка окажется неприемлемо большой, если разделить их компонентом ин-

теграции, таким как шлюз API. Например, в банковском секторе предъявляются весьма строгие требования к скорости обработки платежа. Общее время обработки обычно не должно превышать 100 миллисекунд. За это время нужно многое проверить: уникальна ли оплата? Не обрабатываем ли мы один и тот же платеж повторно? Имеет ли право владелец счета отправить платеж? Нет ли признаков мошенничества при выполнении платежа? Разрешено ли принимающей стороне получить этот платеж? Если бы все приложения или компоненты приложения были разделены через шлюз API, задержка увеличилась бы и предел времени в 100 миллисекунд был бы превышен. Двухточечные соединения решают эту проблему, но для понимания все еще необходимы метаданные с информацией о происхождении. То же относится к проектированию и документации API. Хотя шлюза API нет, области все равно должны выполнять свои обязанности по управлению данными.

Говоря о границах сети и взаимодействиях, нужно упомянуть, что существует еще один шаблон проектирования уровня данных и его компонентов. Он включает в себя развертывание всех общих сервисов централизованно в топологии звездообразной сети.

Топология лучевой сети

Один из подходов более строгого разделения заключается в использовании архитектуры звездообразной сети, которая разделяет области, изолируя сеть. Azure VNet (<https://oreil.ly/VdI5B>), AWS VPC (<https://oreil.ly/4IxIq>) и GCP VPC (<https://oreil.ly/pSKo3>) — все это варианты изолированных сетей в облаке, позволяющих запускать ресурсы в заблокированной виртуальной среде. Благодаря им мы можем развертывать все ресурсы из областей в отдельных логических изолированных сетях — «лучах».

Общие сервисы, включая компоненты уровня данных, находятся в ядре и используются для соединения различных областей (это называется *пирингом*). На рис. 6.8 вы можете видеть, что рабочие нагрузки, требующие подключения к рабочим нагрузкам, действующим на других периферийных серверах, всегда вынуждены строго использовать общие сервисы. Эти сервисы могут расширяться дополнительными средствами управления доступом и безопасностью.

Более сложный вопрос проектирования — следует ли разрешать командам разработчиков периферийных областей развертывать и применять свои компоненты интеграции, размещенные на собственном сервере, — например, шлюзы API и потоковые сервисы. Я вскользь затронул этот вопрос в подразделе «Метаданные и целевая операционная модель» на с. 73 и более подробно остановлюсь на нем в главе 11. Эта операционная модель работает, только если области делают свои данные доступными для обнаружения и дают представление о происхождении

и перемещении. По логике вещей это требует четких стандартов корпоративной отчетности и предоставления метаданных наряду с высококачественными и хорошо организованными данными. Давайте теперь рассмотрим эти стандарты.

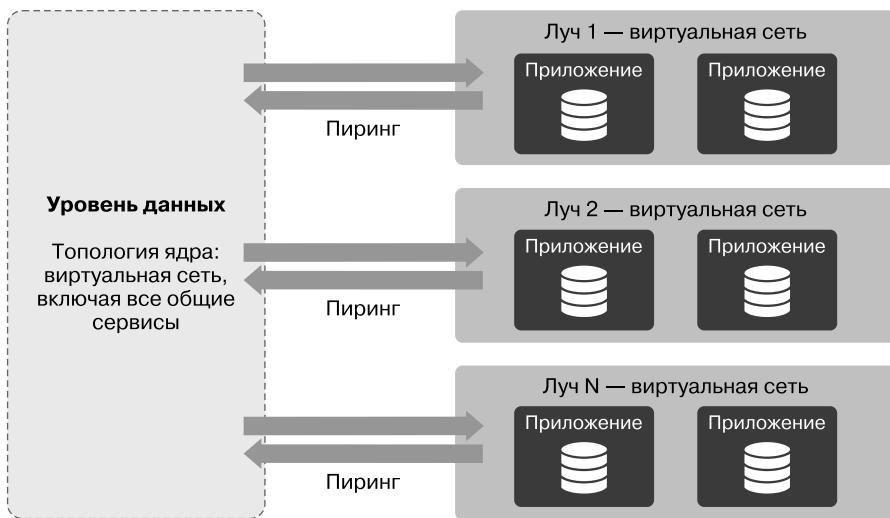


Рис. 6.8. В топологии звездообразной сети области разделяются на уровне сети. В этой модели общие сервисы, такие как сервисы метаданных, размещаются в ядре. Сами области развертываются на периферийных устройствах для поддержания изоляции

Стандарты корпоративных данных

Важнейшая часть перехода к децентрализованной архитектуре данных — понимание того, что интегрирование — это децентрализованное владение, требующее хорошего знания дисциплин. Сравните это с моделью управления разрозненными данными, в которой работают корпоративные хранилища и озера данных. Это больше похоже на владение продуктом DataOps или DevOps, когда люди несут ответственность не только за процесс, но и за качество конечного продукта. Жамак Дехгани (Zhamak Dehghani), главный консультант по технологиям в ThoughtWorks, признает этот сдвиг, отмечая: «Нам необходимо перейти к парадигме, основанной на современной распределенной архитектуре: рассматривать области как первоочередную задачу, использовать платформенное мышление для создания самообслуживаемой инфраструктуры данных и обрабатывать данные как продукт»¹.

¹ Dehghani Z. How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh // Блог Мартина Фаулера, 20 мая 2019 г. <https://martinfowler.com/articles/data-monolith-to-mesh.html>.

Этот переход к распределенному владению данными возможен только при применении широкого спектра стандартов к данным как к продуктам. Без каких-либо корпоративных стандартов распределение и взаимосвязи будут беспорядочными, дезорганизованными и несовместимыми. Эти принципы, подробно обсуждаемые в следующих разделах, сосредоточены на проектировании конечных точек данных, оптимизированных для потребления, четкой отчетности по данным, возможности обнаружения метаданных, семантической согласованности и понимании происхождения и перемещения данных.

Принципы оптимизации потребления

Серьезное отношение к потреблению данных требует рассматривать их как важный актив организации. В рамках этой философии ответственность за качество данных, их представление, детализацию и полноту несут соответствующие владельцы данных (владельцы продуктов). Это также подразумевает управление жизненным циклом интерфейса, включая управление версиями, о чем рассказывалось в предыдущем разделе. Анил Чакраврти (Anil Chakravrthy), бывший генеральный директор Informatica, утверждает (<https://oreil.ly/ZKigS>), что эта философия подхода к данным как к активу коренным образом меняет способы управления данными и их предоставления.

Чтобы избавиться от повторяющейся работы, я предлагаю использовать новый девиз: «Представьте данные правильно один раз и используйте их много раз». Эта философия аналогична *принципу повторного использования сервисов* (<https://oreil.ly/BxpGx>), который пропагандируется в кругах SOA. Данные должны представляться в виде ресурсов многократного использования, чтобы одновременно обслуживать как можно больше клиентов. Они не должны быть ориентированы на каких-либо конкретных потребителей данных.

Цели удобочитаемости и повторного использования влияют на внутреннюю структуру наборов данных, которые будут проходить через уровень данных. Например, бесконечные сложности отношений «родитель — потомок», сильно нормализованные структуры данных или технические их модели усложняют понимание и использование данных потребителями. Поэтому такие данные должны предоставляться с адекватным уровнем детализации, чтобы обслуживать как можно больше потребителей. Данные же, логически связанные друг с другом, должны быть сгруппированы вместе.

Подход к распределению данных, которые могут быть разделены, повторно использованы и оптимизированы для немедленного и простого применения, противоречит многим эталонным архитектурам озера данных. Например, я утверждаю, что RDS нельзя рассматривать как большие хранилища, содержащие огромное количество копий необработанных данных всех исходных

систем в их собственных форматах. Выгрузка необработанных данных (один к одному) со всей их сложностью может принести краткосрочные выгоды. Но она нежизнеспособна в долгосрочной перспективе. Вы же не хотите, чтобы потребители данных сталкивались с необходимостью создавать сложную логику приложения и выполнять массовые объединения или испытывали трудности с интерпретацией данных. Поэтому я призываю команды следовать таким подробным рекомендациям по проектированию интерфейсов.

- Проектируйте конечные точки данных как общие схемы для RDS, API и потоков. Желательно, чтобы все они были созданы из одних и тех же метаданных или кода.
- Представляйте данные с оптимизацией потребления. Это означает, что слишком сильно нормализованные или слишком технические физические модели должны быть преобразованы в более пригодные к многократному использованию и логически сгруппированные наборы данных: *агрегаты предметной области* (<https://oreil.ly/d0STt>). Сложная логика внутри приложения должна быть абстрагирована.
- Единый язык — из предметно-ориентированного проектирования — это язык общения. Это означает, что словарь, язык, структуры данных и соглашения об именах наследуются от области. Данные и контекст должны быть близки к тому, что генерируют операционные и транзакционные системы.
- Поставщики не должны согласовывать свои модели данных с потребностями других областей (единственным исключением для включения бизнес-логики может быть ситуация, когда для интерпретации данных требуются очень специфические знания предметной области).
- Представляйте исчерпывающий набор данных, который может обслуживать максимальное количество областей. Это исключает использование очень специфического формата только для одного потребителя данных.
- Представляйте только те данные, которые действительно интересуют потребителей (а не данные, которые используются только в системе).
- Элементы данных должны быть *атомарными*: то есть их атрибуты не могут быть далее разделены на значимые подкомпоненты. Атомарные элементы данных представляют самый низкий уровень и имеют точное значение или семантику. Области не должны разделять или объединять данные или применять сложную логику для получения правильных значений.
- Используйте согласованные идентификаторы областей. Перекрестные ссылки и отношения внешних ключей должны быть целостными и согласованными

по всему набору данных. Потребители не должны манипулировать ключами для присоединения к наборам данных.

- Данные должны быть отформатированы единообразно во всем наборе. Это подразумевает форматирование представления и соблюдение синтаксических правил оформления, таких как количество знаков после запятой, формы записи и грамматика. Потребляющие области не должны применять дополнительную логику для получения правильных значений.
- Локальные, неуправляемые справочные данные, которые являются непостоянными, должны быть абстрагированы до устойчивого, менее детализированного диапазона значений, который можно легко использовать.
- Предоставьте идентификаторы предприятия, если нужно управлять справочными и основными данными. Этот принцип будет объяснен далее, в главе 9.

Перенос данных на уровень данных с использованием этих рекомендаций по проектированию интерфейсов ускорит процесс создания областей для отдельных сценариев использования. По моему опыту, создание надлежащих конвейеров данных и конечных точек — это больше искусство, чем наука. Поставщики и потребители данных должны согласовать оптимальную их структуру.

Следование этим рекомендациям и взгляд на систему с точки зрения потребителей данных означает, что разблокировка значения требует двух шагов преобразования, как показано на рис. 6.9. Первый шаг ① — данные доставляются поставщиками на уровень данных, такой как RDS. Второй этап преобразования ② выполняется при переносе данных с уровня данных в приложение потребителя. На шаге 1 контекст (семантика) не должен изменяться, в то время как на шаге 2 между уровнем данных и потребителем контекст — и, следовательно, значение данных — изменяется.

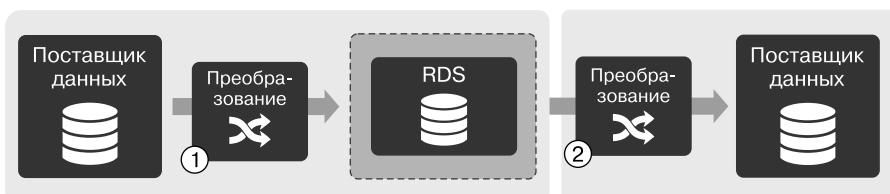


Рис. 6.9. Преобразование данных выполняется как на стороне поставщика, так и на стороне потребителя. Для предоставления данных, оптимизированных для потребления, например, в RDS, поставщик должен сначала преобразовать данные ①. Второй ② этап преобразования выполняется при передаче данных из RDS в приложение потребителя

МОГУТ ЛИ СИСТЕМЫ СОЗДАВАТЬ ИНТЕРФЕЙСЫ, ПРЕДОСТАВЛЯЮЩИЕ НЕОБРАБОТАННЫЕ ДАННЫЕ

Рассмотрите возможность предоставления приложениям необработанных данных и данных, оптимизированных для потребления, одновременно. Такой подход приносит пользу специалистам по данным и потребителям, которым нужен немедленный доступ. Интерфейсы, содержащие необработанные данные, должны быть помечены как частные и предоставляться без гарантий. Их никогда не следует использовать в качестве конвейеров производственных данных.

Определение разделения функций поставщика и потребителя данных — не простая задача, требующая согласованного решения. Иногда трудно сделать выбор, где разместить определенную бизнес-логику и абстракции, потому что некоторые функции явно не вписываются ни в обязанности поставщиков, ни в обязанности потребителей. В таких случаях решение должно приниматься на основе знаний, опыта и здравого смысла.

Самая сложная бизнес-логика связана с обслуживанием справочных данных, которые должны отображаться в справочные данные на другой стороне. Если справочные значения детализированы и часто меняются, потребителям будет сложно угадаться за этими изменениями. Каждое изменение, внесенное в справочные данные, повлияет на потребителей данных. Поэтому поставщики и потребители должны согласовать правильный уровень детализации. Поставщикам данных может потребоваться абстрагировать или свести детализацию локальных справочных данных к более общей форме, не зависящей от потребителя, которая более стабильна и не меняется так часто.

Чтобы поддержать команды разработчиков областей в их стремлении оптимизировать данные для потребления, я рекомендую создать экспертно-консультационный центр для выработки проектных решений. Этот центр тоже должен разрабатывать четкую документацию, подробные описания вариантов использования и примеры кода. Примером может служить документация-руководство по API Zalando (<https://oreil.ly/bqGm5>) — это открытый документ, который направляет разработчиков и облегчает им совместную деятельность.

Методологии проектирования, обсуждаемые в этом разделе, требуют от команд предметных областей большой дисциплины и внимания к качеству, а также четкого определения права собственности.

Возможность обнаружения метаданных

Хотя платформы поддержки уровней данных могут предоставляться централизованно или формироваться разными сторонами самостоятельно, право собственности и ответственность за конечные точки данных всегда остаются

за областью-поставщиком. Это означает, что к одним и тем же данным, доступным через несколько конечных точек одновременно, применяется одно и то же право собственности. Например, когда данные одновременно распределяются с использованием RDS, API и конечных точек потоковой передачи, все они принадлежат одной области. На рис. 6.10 показано, как все конечные точки могут принадлежать одной области (одному и тому же ограниченному контексту) и вести обратно к одному владельцу.

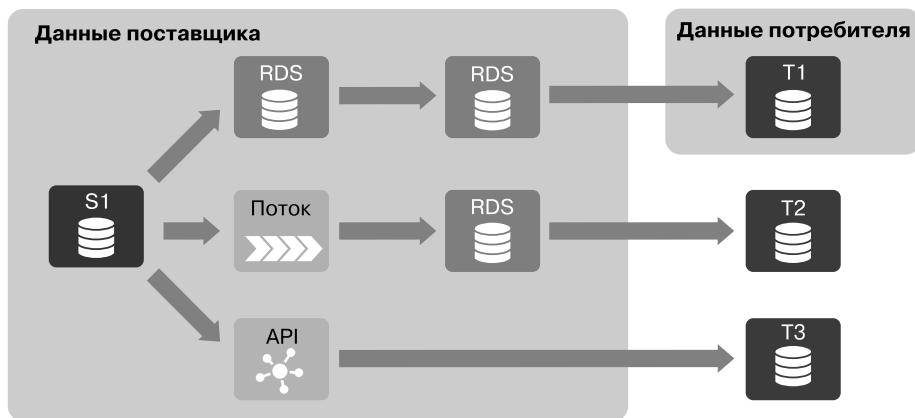


Рис. 6.10. Когда данные распределяются одновременно с использованием нескольких шаблонов интеграции, владелец остается ответственным за них

Владение данными и управление ими — важные аспекты обеспечения прозрачности и доверия к данным. Мы обсудим это более подробно в главе 7, но здесь я хочу подчеркнуть, как сделать метаданные о владении данными, золотые наборы данных и интерфейсы доступными для всех областей. Для этого я рекомендую использовать модель метаданных предприятия.

На рис. 6.11 можно увидеть, как владельцы данных связаны с золотыми источниками, золотыми наборами и элементами данных. Золотые источники, как вы уже знаете, — это приложения, в которых обрабатываются надежные данные; золотые наборы данных — это независимые от технологий представления, используемые для классификации данных и связывания их с владельцами и элементами данных; а золотые элементы данных — это атомарные единицы информации, которые действуют как клей, связывающий физические данные, интерфейс и метаданные моделирования данных. Эти метаданные остаются абстрактными для обеспечения гибкости.

Синим цветом показаны физические представления данных. Эти данные используются для чтения, поиска или записи. Их можно рассматривать как

метаданные интерфейса, так как они описывают представление данных или их распределение из золотых источников. Хотя эта модель может работать с любым типом интерфейса, здесь я ее упростил, добавив только шаблон пакетной обработки в автономном режиме с использованием плоских файлов. Эти файлы принадлежат интерфейсам, а несколько интерфейсов принадлежат приложению. Каждый интерфейс также должен иметь версии для развития схемы и совместимости. На детальном уровне атрибуты данных соответствуют файлам. Атрибуты должны содержать много деталей, например, являются ли данные уникальными, допускают ли они значение NULL, индексированы ли они и т. д. Склейивание физических атрибутов и элементов данных вместе осуществляется через промежуточную таблицу: `ElementAttribute`.

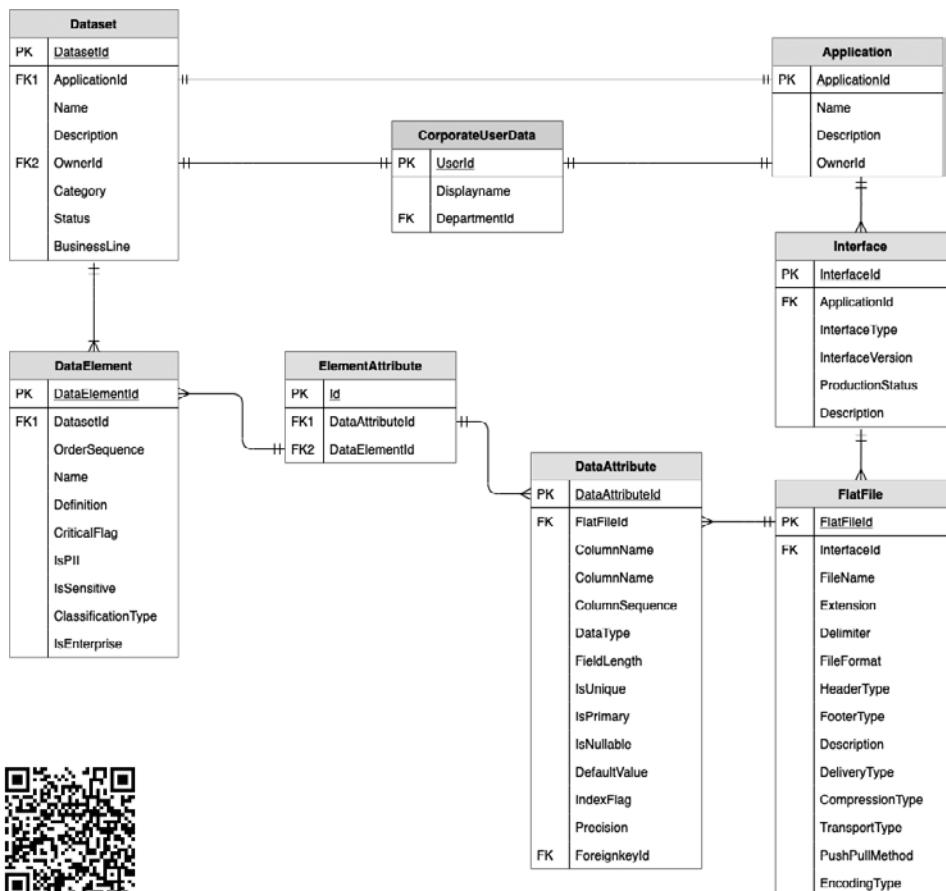


Рис. 6.11. Метаданные управления (выделены зеленым цветом) используются для классификации наборов данных и элементов данных, принадлежащих определенным владельцам



Почему желательно абстрагировать информацию о владении, а не связывать ее напрямую с физическими атрибутами данных? Чтобы избежать образования тесных связей. Если изменяются владельцы, субъекты бизнеса или классификации, все соответствующие интерфейсы тоже должны измениться. Разделяя и связывая элементы золотого набора данных, вы делаете архитектуру более гибкой.

Одна из причин *не* использовать концептуальную модель данных в качестве нашей модели управления в том, что концептуальные модели данных обычно богаче. Они могут представлять больше информации и давать больше контекста, чем фактическая реализация. Еще одна причина отказа от использования концептуальных моделей состоит в том, что концептуальные модели могут обладать более динамическими свойствами и отношениями с другими предметными областями. Подробнее об этом рассказывается в главе 10.

Для поддержки возможности обнаружения рекомендую сделать все метаданные, включая соответствующие интерфейсы и схемы, легкодоступными, например, через API.

Пример 6.1 показывает, как может выглядеть запрос к такому сервису. Обратите внимание, что каждая область, использующая собственный ограниченный контекст, должна иметь возможность регистрировать свои наборы данных и соответствующие им URI. Так, помимо предоставления сервисов обнаружения, архитектура должна помочь областям самостоятельно регистрировать свои наборы данных и интерфейсы.

Пример 6.1. Надуманный пример списка золотых источников в формате JSON, возвращаемый сервисом обнаружения

```
{  
    "ApplicationId": 23,  
    "ApplicationName": "CRM",  
    "ApplicationDescription": "Retail Customer Relationship Management",  
    "ApplicationOwnerId": 126,  
    "UniqueDataSets": {  
        "DatasetId": 2045,  
        "DatasetName": "CRM Customer Data",  
        "DataOwnerId": 97,  
        "Status": "Production",  
        "DataElements": [  
            {  
                "DataElementId": 987,  
                "OrderSequence": 1,  
                "Name": "UniqueIdentifier",  
                "Description": "Unique customer identifier"  
            },  
            {  
                "DataElementId": 988,  
                "OrderSequence": 2,  
            }  
        ]  
    }  
}
```

```
        "Name": "CustomerFirstname",
        "Description": "Customer first name",
        "IsSensitive": 1
    },
    {
        "DataElementId": 989,
        "OrderSequence": 3,
        "Name": "CustomerLastname",
        "Description": "Customer last name",
        "IsSensitive": 1
    }
]
},
"InterfaceLocations": [
{
    "InterfaceId": 535,
    "Url": "adl://data.corp/data_exports/CRM/crm.csv",
    "SchemaVersion": "v1.0",
    "DataRequestType": "dcat:FlatFile",
    "Attributes": [
        {
            "DataElementId": 987,
            "name": "ID",
            "type": "int"
        },
        {
            "DataElementId": 988,
            "name": "CUST_FIRSTNAME",
            "type": "string"
        },
        {
            "DataElementId": 989,
            "name": "CUST_LASTNAME",
            "type": "string"
        }
    ]
}
]
```

Семантическая согласованность

При реализации возможности обнаружения вам также понадобится защитить согласованность данных. Для этого нужно, чтобы все области полностью задокументировали данные своих приложений и модели интерфейсов. Они должны быть подробно описаны, но, что наиболее важно, все атрибуты данных *должны* быть связаны с элементами золотого набора данных.

Важную роль играет также смысловая передача контекста, иначе может возникнуть семантическая путаница в описаниях данных, принадлежащих разным

областям. Похожие термины могут иметь совершенно разное значение в разных областях. Вам понадобится соотнести каждый элемент данных с правильной областью и понять, что он означает, откуда возник и где физически хранится. В этом разделе я покажу, как связать все эти разные модели вместе.

Прежде чем переходить к решению, давайте быстро освежим память с помощью иллюстрации. В главе 1 я рассказал об этапах проектирования: концептуальном, логическом и физическом. Помните, что концептуальная модель данных представляет сущности бизнеса; логическая модель — структуру базы данных приложения; а физическая описывает фактически реализованный проект. В идеальной ситуации все три модели связаны между собой, поэтому пользователи понимают, как бизнес-концепции претворяются в жизнь в реальных условиях.

При проектировании подхода с тремя моделями на уровень данных, как показано на рис. 6.12, концептуальные модели представляют первые (верхние) модели. На стороне поставщика, слева, концептуальные модели данных области остаются прочно связанными. Контекст по мере распространения данных не меняется. Единственное отличие состоит в том, что данные предметной области, предоставляемые через уровень данных, имеют другое представление и обычно являются подмножеством данных исходной системы или приложения.



Рис. 6.12. В идеале все атрибуты логической или физической модели данных приложения связаны. Поскольку данные на уровне данных находятся на стороне поставщика, их атрибуты также будут связаны с концептуальной моделью поставщика

Логические модели, которые все еще находятся на стороне поставщика, происходят из концептуальных моделей. Они представляют проекты баз данных и модели интерфейсов. Каждой настройке дается соответствие, которое следует хранить в средстве моделирования или проектирования данных.



Логические модели обычно имеют смысл только при разработке проектов баз данных, не зависящих от приложений, например технологий реляционных баз данных. Однако в последнее время логическое моделирование стало нецелесообразным, потому что в большинстве современных моделей используются либо схемы NoSQL, либо денормализованные структуры данных. Поэтому на практике многие инженеры сейчас обходят этап логического моделирования.

Модели физических данных приложений и модели интерфейсов, которые описывают уже реализованные проекты, должны аккуратно сохраняться в репозиториях метаданных. Однако здесь есть свои сложности, потому что области могут реализовывать несколько интерфейсов для своих приложений. Одни и те же данные могут иметь множество представлений, каждое из которых требует, чтобы области отображали каждый атрибут своих интерфейсов в соответствующие элементы золотого набора данных. Ожидается, что, экспортируя данные, области будут следовать принципам повторной оптимизации.

На стороне потребителя, как ожидается, тоже будут существовать все модели. Разница лишь в отсутствии моделей интерфейса, но данные будут преобразовываться и интегрироваться. Концептуальные модели данных также отличаются друг от друга из-за изменения контекста. Эти концептуальные модели являются основой для логических и физических моделей данных приложения. Чтобы иметь полное представление обо всем процессе, храните и связывайте все метаданные вместе.

Нет простого решения для соединения и проверки взаимоотношений между различными моделями в едином репозитории. Чтобы реализовать весь потенциал, необходимо разработать инструменты, а также объединить и интегрировать метаданные. Вы можете начать с каталога, репозитория или портала документации и, двигаясь вперед, стремиться к согласованности и пониманию общих черт и различий между контекстами во всех архитектурах. Для этого нужно будет сблизить репозитории моделей данных.

Решение, которое я рекомендую, заключается в том, чтобы собрать и сохранить все бизнес-концепции и взаимосвязи из концептуальных моделей данных централизованно в форме онтологии, связать их с элементами золотых наборов данных, которые могут храниться в другом репозитории, а за-

тем, наконец, проверить все на соответствие метаданным (схемам) моделей интерфейса.



В зависимости от синтаксиса вашего интерфейса можно реализовать автоматический перевод терминов из написания с пробелами в змеиную (<https://oreil.ly/XbwPi>) или верблюжью нотацию (https://oreil.ly/7M_Zy). Не все базы данных поддерживают пробелы в именах столбцов.

Самый элегантный подход к соединению всех метаданных — потребовать от поставщиков данных предоставить схемы интерфейсов, включая дополнительные атрибуты метаданных, для ссылки на элементы золотого набора данных. При таком подходе метаданные доставляются вместе с данными или инкапсулируются в них. В качестве альтернативы попросите поставщиков выгрузить свои схемы и метаданные интерфейсов в центральный репозиторий и связать все атрибуты данных с элементами золотого набора данных.

Другой подход, требующий дополнительных усилий, — разработка небольшого веб-сайта или приложения, визуализирующего все схемы БД на уровне данных и интерфейсы конечных точек и требующего, чтобы поставщики связали атрибуты данных с соответствующими элементами золотого набора.

Разберем конкретный пример. Рассмотрим `household` (рус. «домовладение») — термин, который используется во многих сферах бизнеса. Он будет храниться как бизнес-объект в концептуальной модели данных (рис. 6.13, *справа*), а также существовать как элемент данных в репозитории золотого набора данных. Термин должен быть сопоставлен с бизнес-объектом, чтобы показать, как контекст соотносится с данными и их принадлежностью. Следующее упражнение — сопоставить `household` с атрибутами данных интерфейсов.



На рис. 6.13 показано, что каждый бизнес-термин всегда связан с одним элементом данных. Но так бывает не всегда. Чаще один бизнес-термин связан с несколькими элементами данных, за которыми может быть закреплено несколько физических имен. Кроме того, допускается различие представлений сущностей концептуальной модели данных и элементов золотого набора данных. Например, термины, используемые в концептуальной модели, могут иметь другое представление — более длинные имена и т. д.

Первый шаг к достижению согласованности между моделями — проверка отношений между бизнес-объектами и элементами золотого набора данных. Второй шаг — проверка моделей интерфейса или физических данных. Метаданные на этом шаге должны соответствовать и проверяться на соответствие метаданным из репозитория золотого источника.

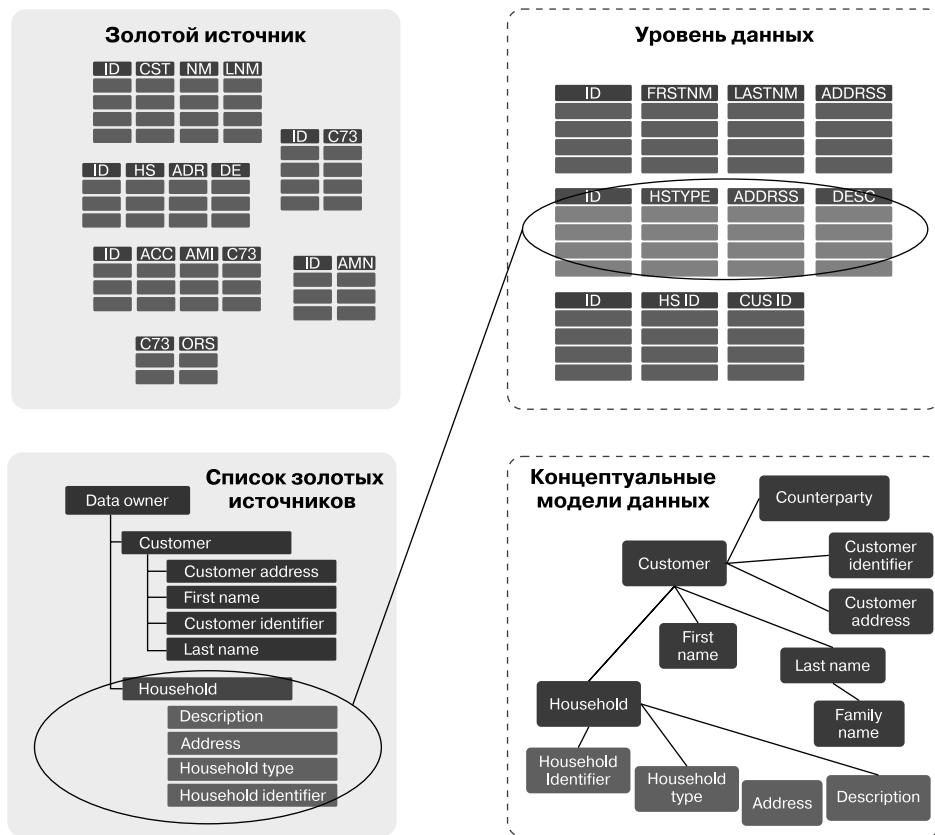


Рис. 6.13. Наборы данных могут быть связаны как с бизнес-объектами, так и с объектами данных. Если все сделано правильно, взаимосвязь демонстрирует возникновение требований к данным для развертывания физических моделей на уровне метаданных

В зависимости от требований к контролю и управлению данными вы можете проверить охват интерфейсов на уровне данных. Если ни один из атрибутов метаданных интерфейса не соответствует элементам золотого набора данных, то, возможно, стоит запретить поставщикам публиковать такие интерфейсы. Кроме того, вы можете составить отчет об объеме покрытия, чтобы показать, какие области выполнили свою работу хорошо или плохо.

Предоставление соответствующих метаданных

Значительная часть усилий по созданию масштабируемой архитектуры уйдет на то, чтобы сделать ее управляемой с помощью метаданных. Чтобы данные стали доступны для обнаружения и повторного использования, а также чтобы можно

было применить правильные ограничения, необходимо, чтобы метаданные были встроены в данные. То же относится и к возможностям взаимодействий. Чтобы упростить передачу данных между различными системами и избежать блокировки, также необходимо опубликовать метаданные и сделать их доступными для всех сторон.

Большинство требований к метаданным будут обсуждаться в главе 10, но для данных, которые будут публиковаться через архитектуры, важно на этапе реализации визуализировать унифицированную метамодель с критическими метаданными, которые должны быть опубликованы или предоставлены вместе с данными: схемы, идентификаторы приложений, идентификаторы областей, идентификаторы бизнес-возможностей, идентификаторы владения данными, классификации целей, связь с элементами золотого набора данных и т. д. Для удобства обнаружения все должно быть объединено в каталоги, реестры или порталы разработчиков.

Происхождение и перемещение данных

Происхождение — жизненно важный аспект управления данными. Это палочка-выручалочка, которая помогает проследить, откуда произошли данные, как они собирались, как изменились и как будут потребляться в дальнейшем. Информация о происхождении помогает проследить движение потоков данных по предприятию. Потребность в этой информации обусловлена соблюдением требований, нормативными актами, конфиденциальностью, этикой, а также необходимостью обеспечить воспроизводимость и прозрачность моделей продвинутой аналитики.

Чтобы последовательно фиксировать происхождение, следуйте принципу не-преложного использования уровня данных. То есть при потреблении, преобразовании на стороне потребителя и повторном распределении обязательно используйте уровень данных. Это позволит проследить «перемещение» данных и встроить их происхождение в архитектуру.

Происхождение на уровне данных, как показано на рис. 6.14, либо генерируется автоматически каждый раз, когда данные пересекают его, либо создается и доставляется областями вручную. Оно хранится в центральном репозитории происхождения¹. При правильном проектировании компоненты интеграции во всех архитектурах будут автоматически принимать информацию о происхождении. Поскольку еще не существует платформы, способной генерировать все метадан-

¹ Этот репозиторий может применять индивидуальное решение с графовой базой данных и сетевой визуализацией. Если вам нужно получать эти данные в реальном времени, я предлагаю использовать тему потоковой передачи или REST API для публикации происхождения.

ные о происхождении для всех шаблонов интеграции, я рекомендую спроектировать и создать центральный репозиторий для метаданных о происхождении самостоятельно. В качестве альтернативы вы можете приобрести коммерческий продукт, реализующий серверную часть такого репозитория, но существует высокая вероятность, что вам все равно придется ее настраивать, поскольку требования к интеграции и инструменты различаются для разных областей.

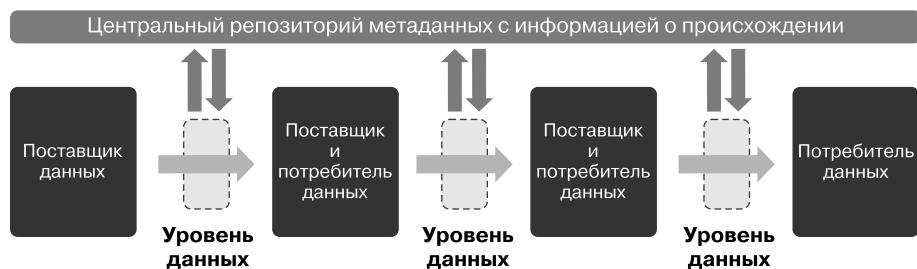


Рис. 6.14. Информация о происхождении позволяет создать общую схему данных. Чтобы ее иметь, необходимо фиксировать движение данных каждый раз, когда они пересекают уровень данных

Минимальное требование к информации о происхождении — перечень приложений, участвовавших в преобразовании данных, и какие архитектуры интеграции, платформы и возможности интеграции использовались для этого. Исходя из этих требований, в качестве начальной модели можно использовать конструкцию, представленную в табл. 6.2 и на рис. 6.15.

Таблица 6.2. Пример структуры метаданных о происхождении

AgreementId	UniqueHashKey	SourceId	TargetId	SourceSchemaId	TargetSchemaId	IntegrationType
1	H7Q9K1L	345	85			API
2	5TJ5JMN	346	861	TABLE01	TABLE02	RDS
3	09HB69M	532	103	TABLE01/FIELD01	TABLE01/FIELD02	RDS
				TABLE01/FIELD01	TABLE02/FIELD01	
4	12WRTMD	98	17	TABLE01/FIELD01	TABLE01/FIELD02	RDS
				TABLE01/FIELD01	TABLE01	
				TABLE02/FIELD02		

В табл. 6.2 также есть *хеш-ключ* — уникальный идентификатор, который области могут использовать для поддержания происхождения. Если по какой-либо

причине предлагаемые возможности централизованной интеграции не смогут сгенерировать информацию о происхождении, области воссоздадут и доставят ее самостоятельно, используя уникальный хеш-ключ. При таком подходе вы не потеряете понимание распределения данных.

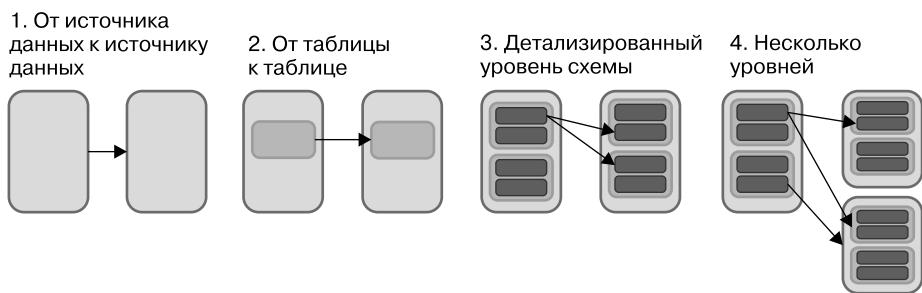


Рис. 6.15. Имея метаданные с информацией о происхождении, можно визуально представить и выяснить движение данных от источника к месту назначения на разных уровнях: от источника к источнику, от таблицы к таблице, на уровне поля или в их комбинациях

Также можно предусмотреть создание уникального идентификатора происхождения для каждой строки, события или вызова API и привязать его к фактическим данным. RDS, например, могут расширять таблицы дополнительными столбцами с *универсальными уникальными идентификаторами* (universally unique identifier, UUID). Каждая уникальная строка получает уникальный идентификатор, который позволяет точно отслеживать все данные.

Стандарты корпоративных данных важны для обеспечения масштабируемости распределения данных в интегрированной среде. Управление и стандарты необходимо установить в централизованном порядке. При этом отдельно должны располагаться команды областей, которые могут быть полностью независимыми. Все должно работать, пока команды соблюдают все принципы.

Эталонная архитектура

Соберем все вместе. Вы узнали, что приложения могут предоставлять свои данные через разные конечные точки и что метаданные важно фиксировать. В предыдущих главах вы также узнали, что разные архитектуры могут работать вместе. Например, событийно-ориентированная архитектура может использоваться для ввода данных в RDS; API можно развертывать непосредственно поверх RDS. Объединив архитектуры с ролями «золотой источник» и «хранилище

данных области» (domain data store, DDS), вы получите первое представление об общей архитектуре высокого уровня. Посмотрите на рис. 6.16, чтобы узнать, что находится внутри.

Посередине, между поставщиками и потребителями данных, мы видим уровень данных, включающий различные компоненты интеграции. Три архитектуры интеграции из предыдущих глав не выделяются в общую архитектуру сами по себе, но дополняют друг друга и тесно взаимодействуют между собой. Слева находятся поставщики данных с их *системами золотых источников*. Именно отсюда берутся данные и начинают распространение золотые наборы данных. Они включают как операционные, так и аналитические приложения. Справа изображены потребители данных, DDS. Они разрабатываются индивидуально на основе сценария использования, который также может быть операционным или аналитическим. Из-за того что варианты использования различаются и настраиваются, модели данных также должны быть конкретными. Шаблоны и шаги интеграции могут быть разными: от одного шаблона запрос/ответ в реальном времени до нескольких этапов ETL, включая очистку данных, обогащение и т. д. Все это станет понятным в главе 8.

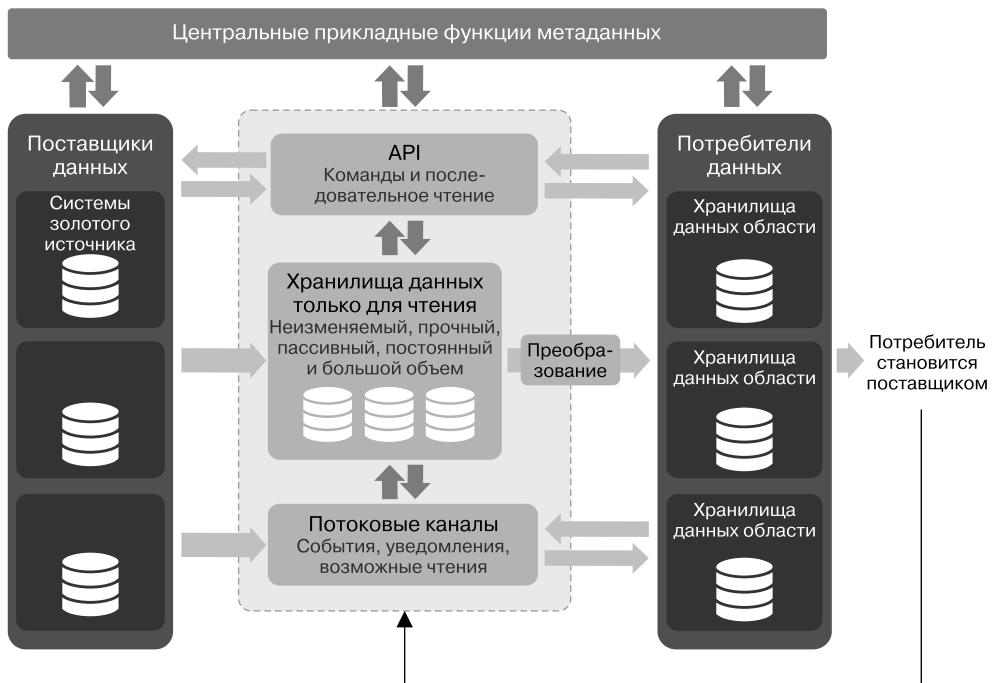


Рис. 6.16. Эталонная архитектура, включающая все архитектуры с ролями «золотой источник» и «хранилище данных области»

Вверху находится еще один уровень: *уровень метаданных*, который будет подробно обсуждаться в главе 10. Это абстрактный уровень, содержащий все архитектурные блоки, которые полагаются на метаданные. Вы уже видели некоторые из них, когда мы обсуждали архитектуры RDS, API и потоковой передачи. Они помогают обеспечить единообразное представление всех данных и понимание семантики, перемещения данных, владения и т. д.

Потребители данных также могут стать поставщиками. Приложения, как описано в главе 2, могут потреблять, интегрировать и создавать новые данные. В такой ситуации только вновь созданные золотые наборы данных могут передаваться на уровень данных. На рис. 6.16 это поясняется стрелкой справа, направленной вниз. Когда потребители становятся поставщиками, они должны придерживаться тех же принципов, что и любые другие поставщики данных.

Как в эту архитектуру вписываются хранилища и озера данных? Они могут располагаться по обе стороны как поставщики или потребители данных. Это же касается и операционных систем. Скорее всего, большинство из них будет находиться на стороне поставщика. Но так как они тоже могут интегрироваться и потреблять данные, они могут быть и на стороне потребителя.

Итоги главы

Масштабируемая архитектура отличается от многих других, ведь она объединяет три архитектуры интеграции. Она не зависит от технологий, комплексна, ориентирована на будущее и не исключает использования приложений любого типа. Такая архитектура поддерживает не только микросервисы с Kubernetes, Hadoop или Kafka. Это общий пример проектирования, который упрощает использование многих технологий для последовательного обмена данными и их распространения.

По мере того как мы уходим от проблем связаннысти, метаданные становятся kleem, который скрепляет все вместе. Они дают представление обо всех потоках данных. Благодаря принципам разделения, описанным в этой главе, архитектура остается гибкой. Каждая область может развиваться, потреблять и предоставлять данные независимо и со своей скоростью. Единственной зависимостью предметных областей являются уровни данных. Для слаженной работы все области должны предоставлять данные в оптимизированном для потребления и многократного использования виде. В этом случае вы сможете извлечь данные только один раз и использовать их многократно для разных целей, а неуклонное следование принципам пересечения сетей позволяет избежать повторного распространения одних и тех же данных.

У модели предметно-ориентированного проектирования есть недостаток, заключающийся в том, что группы-потребители должны понимать различные контексты областей. Интеграция потребует больше усилий, но улучшенная гибкость и удобные принципы использования потребляемых данных делают эти усилия не напрасными. Когда мы обсудим управление основными данными в главе 9, вы узнаете, что некоторое корпоративное единство будет возвращено в архитектуру путем обработки, обслуживания и повторной публикации данных.

В следующих главах архитектура будет расширена за счет большего количества областей и дисциплин управления данными для превращения данных в ценность, включая управление данными, безопасность данных, управление метаданными и основными данными, а также DDS.

ГЛАВА 7

Управление данными и их безопасность

Одно из основных свойств масштабируемой архитектуры — переход от владения на основе приложений к владению на основе данных. Это требует, чтобы управление данными и их безопасность были глубоко встроены в способы взаимодействия и использования данных приложениями и пользователями.

Почему управление данными и их безопасность обсуждаются в этой главе вместе? Потому что они пересекаются! Несмотря на распространенное мнение, что управление данными и их безопасность — разные дисциплины, они работают вместе и дополняют друг друга. Они имеют общую цель: управление данными определяет, что они представляют и для чего могут быть использованы, а затем безопасность данных гарантирует, что только авторизованные стороны могут получить доступ к данным (в соответствии с тем, для чего данные могут быть использованы).

Для соединения областей и единообразного применения аутентификации и авторизации во всех архитектурах требуется много метаданных. Необходимо назначить владельцев данных, установить классификации и поддерживать соглашения между сторонами. Информация, как и ожидалось, должна храниться централизованно в единой модели. Как вы понимаете, большая часть главы посвящена этому.

Управление данными

Управление данными, как мы говорили ранее, состоит из действий по реализации и обеспечению полномочий и контроля над управлением данными, включая соответствующие активы.

Зачем нужно управление данными? Это правильный вопрос, потому что в небольших компаниях или стартапах в управлении обычно нет необходимости

или оно проводится неявно, как часть повседневной деятельности. Большинство приложений и данных в небольших компаниях обычно принадлежат небольшому количеству людей и ими же управляются, но при этом данными пользуются все! Объемы и разновидности данных относительно невелики, поэтому их поиск или получение ответов на вопросы занимает мало времени. Часто достаточно просто расспросить коллег, а если они не знают ответа, то он наверняка находится где-то рядом.

Проблемы возникают, когда компании начинают расти. Время в пути между отделами увеличивается, все больше людей имеют разные точки ответственности, знания разбросаны, решения принимаются дольше, обязанности становятся неясными и т. д. Организация, отчетность и прозрачность становятся все более актуальными по мере увеличения потребления и использования данных. Необходимость управления данными все более очевидна.

Кроме того, компании сталкиваются с новыми нормативными актами, такими как европейский Общий регламент по защите данных (General Data Protection Regulation, GDPR) и Закон штата Калифорния о защите конфиденциальности потребителей (California Consumer Privacy Act, CCPA). Эти правила требуют полного контроля над использованием данных, понимания распределения и четкого определения обязанностей. Эти законы также требуют, чтобы вы идентифицировали и четко документировали, где были сохранены данные, откуда они возникли, для чего и как используются. Соответственно, получение этой информации при работе с тысячами приложений и баз данных — сложная задача.

Управление данными на высоком уровне охватывает пять ключевых измерений, таких как организация (культура), процессы, технологии (приложения), люди и данные.

- *Организация.* Измерение организации — это четкое определение ролей и обязанностей организации — владельцев данных, владельцев приложений и пользователей данных.
- *Процесс.* Измерение процесса — это то, как процессы должны контролироваться и проверяться. Процессы управления качеством и безопасностью данных обычно являются критически важными, но существуют также процессы управления справочными и основными данными, жизненным циклом данных, бизнес-исследованиями и аналитикой, а также моделями данных, включая поддержку определений данных.
- *Технологии.* Технологическое измерение в основном сосредоточено на стандартизации интерфейсов, инструментов и фреймворков, которые позволяют контролировать управление данными.

- *Люди.* В человеческом измерении основное внимание уделяется человеческим аспектам, включая этические компромиссы, юридические соображения, предубеждения и социальные и экономические соображения.
- *Данные.* Измерение данных фокусируется на них самих: классификации, определения, происхождение и т. д.

В следующих разделах мы рассмотрим каждое из этих измерений более подробно и обсудим связь между ними и то, как все это отображается в общей архитектуре.

Организация: роли в управлении данными

Измерение организации сосредоточено на ролях и четко определенных обязанностях владельцев и пользователей данных и приложений. До сих пор мы в основном использовали абстрактные концепции, такие как поставщики и потребители данных. Но в этой главе пришло время разбить их на более конкретные роли. Каждая организация структурирует и называет эти роли по-своему, поэтому названия или ключевые действия в общей структуре управления данными могут различаться для разных предприятий. Вот общий обзор самых распространенных ролей, обязанностей и подотчетности организации, занимающейся управлением данными.

- *Владелец данных.* Владелец данных, иногда называемый *доверенным лицом данных* или *владельцем процесса*, — это отдельный сотрудник в организации, который несет ответственность за данные и надлежащее управление соответствующими метаданными¹, включающими в себя качество, определение и классификацию данных; цели, для которых могут использоваться данные, метки и т. д. Подотчетность в распределенной экосистеме не ограничивается только корпоративными данными. Владельцы данных также могут нести ответственность за внешние и открытые данные.
- *Пользователь данных.* Пользователь данных — это отдельный сотрудник в организации, который намеревается использовать данные для определенной цели и несет ответственность за определение требований.
- *Создатель данных.* Создатель данных — это внутренняя или внешняя сторона, которая создает данные по согласованию с владельцем данных.
- *Потребитель данных.* Потребитель данных — это внутренняя или внешняя сторона, которая использует данные в соответствии с намерениями владельца данных и/или пользователя данных.

¹ Владелец процесса также может быть владельцем данных, потому что во многих случаях он также владеет данными.

- *Владелец приложения.* Владелец приложения, иногда называемый *хранителем данных*, поддерживает ядро приложения и его интерфейсы. Владелец приложения несет ответственность за предоставление, функционирование и использование сервисов для бизнеса, а также за поддержание информации о приложении и контроль доступа. Владелец приложения может быть внутренним или внешним.
- *Наблюдатель.* Наблюдатель за данными следит за соблюдением политик и стандартов данных. Часто это профильные эксперты по определенному типу данных.

Все эти роли обычно объединяются в хорошо продуманную инфраструктуру управления данными, которая устанавливает руководящие принципы, правила, действия, роли и обязанности, нужные для корректного управления данными. Эта инфраструктура обычно также включает структуру, в которой должны работать участники: *группу управления данными* или руководящий орган.

Члены этой группы совместно устанавливают стандарты и единообразные политики управления данными. Кроме того, они определяют действия и процедуры, которым должны следовать различные роли — например, наблюдатели данных. В руководящий орган обычно входят и другие руководители и представители — архитекторы данных или менеджеры из других бизнес-отделов. Сам орган в целом напрямую подчиняется управляющему данными. И последнее замечание, касающееся основных действующих лиц и различных ролей: на крупных предприятиях роли, выделенные ранее, обычно являются объединенными — их выполняют люди в децентрализованных областях.

Когда орган управления данными назначает разные роли, он должен подробно описать соответствующие обязанности и задачи. Владельцы и пользователи данных, владельцы приложений и наблюдатели данных должны осознавать свои роли и роли своих коллег. Эти наборы задач также включают в себя ряд процессов (о которых я расскажу позже). Общие обязанности обычно изложены в матрице RACI (<https://oreil.ly/KgNs7>)¹, которая отображает все различные задачи, роли и обязанности, а также тех, кто несет ответственность.

Для владения и обмена данными, а также их использования между различными приложениями все роли должны эффективно взаимодействовать на всех уровнях. Они должны быть структурированы в согласованную модель с ключевыми принципами проектирования. Начнем с иллюстрации того, как это будет работать.

На рис. 7.1 вы видите, что различные роли работают вместе на двух уровнях. Уровень использования данных ориентирован на их создание и потребление.

¹ <https://secretmag.ru/enciklopediya/chto-takoe-matrica-raci-obyasnyaem-prostymi-slovami.htm>. — Примеч. ред.

Его основная цель — четко определить данные, взять на себя ответственность и согласовать их цель. Сюда входят заинтересованные стороны бизнеса: владельцы, создатели, пользователи и потребители данных. На этом уровне устанавливаются функциональные требования: соглашения о требованиях и характеристиках использования данных. Они, как описано в подразделе «Контракты на поставку данных и соглашения о совместном их использовании» на с. 63, фиксируются посредством соглашений о совместном использовании данных.

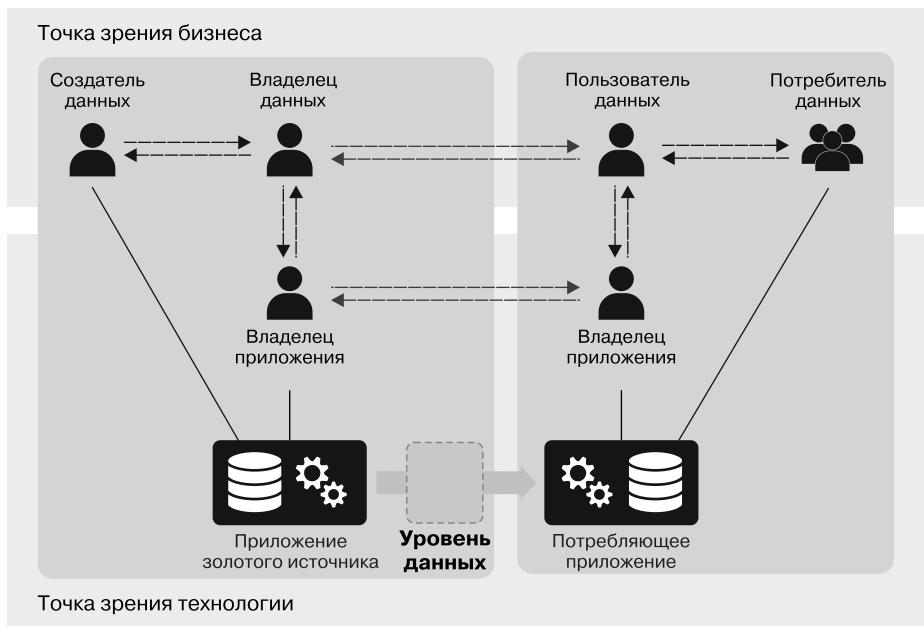


Рис. 7.1. В инфраструктуре управления данными в масштабируемой архитектуре проводится различие между владением данными и владением приложениями

Технический уровень фокусируется на технологических аспектах приложений, таких как интерфейсы между приложениями, соглашения об уровне обслуживания, протоколы и управление версиями. Эти свойства фиксируются с помощью контрактов на поставку данных. Стрелки между разными ролями указывают на то, что заинтересованные стороны должны работать в тесном взаимодействии как в бизнесе, так и в ИТ. Например, если данные недоступны на уровне данных, ожидается, что владельцы проверят, можно ли их сделать доступными. Если это так, они должны обратиться к соответствующим владельцам приложений, чтобы они начали создавать интерфейсы и инициировали процесс интеграции данных.

В рамках системы управления данными наблюдатель данных не упоминается. Поскольку наборы данных принадлежат владельцам данных, маловероятно, что наблюдатели будут участвовать в таких действиях, как определение данных, улучшение качества и ответы на вопросы. Тем не менее владелец данных может делегировать ответственность наблюдателю.

Процессы: деятельность по управлению данными

Орган управления должен обеспечивать эффективное управление данными в организации. Обычно это достигается с помощью хорошо сформулированных процессов и процедур и выполняется разными ролями. Так, значительная часть работы по управлению данными включает определение принципов, политик, правил и стандартов, которые соответствуют целям предприятия и его склонности к риску.

Рассмотрим некоторые из самых важных.

- Распределение ролей и обязанностей по процессам управления данными.
- Выявление потенциальных источников и владельцев данных и добавление их всех в центральный репозиторий.
- Определение политик для правильной обработки данных в соответствии с применимыми нормативными актами, такими как GDPR для обработки информации, позволяющей установить личность (*personally identifiable information*, ПII), и постановления, принятые Базельским комитетом по банковскому надзору для финансовых учреждений.
- Определение уровней качества данных при создании и распространении, включая допустимые уровни погрешностей, а также распределение ответственности за качество данных и мониторинг последующих действий.
- Определение степени детализации информации о происхождении данных и типов приложений, необходимых для ее доставки.
- Настройка центральной схемы классификации и репозитория для использования, добавления тэгов, маркировки и защиты данных.
- Изучение влияния новых правил на управление данными.
- Согласование с архитектурой предприятия, чтобы политики были глубоко встроены в архитектуру.
- Определение того, какие данные необходимо обрабатывать на центральном уровне.

- Настройка политик управления жизненным циклом данных, в том числе для других информационных активов, таких как отчеты и информационные панели.
- Определение требований к идентификации схемы и стандартов протокола для распространенных форматов данных, таких как XML, CSV и JSON.
- Определение правил именования метаданных и типов данных, а также соглашений о форме записи, таких как змеиная или верблюжья нотация.
- Обеспечение правильной доставки метаданных командами областей.
- Формирование культуры осведомленности о данных, включая этику и использование машинного обучения в определенных контекстах.
- Мониторинг и инструктаж пользователей для обеспечения правильного обслуживания моделей данных.
- Разработка методологий и передовых практик моделирования данных, разработки онтологий, ведения бизнес-глоссариев и т. д.
- Ведение каталога данных и обеспечение подробного описания всех информационных активов.
- Выявление отклонений от правил управления данными и устранение связанных рисков.
- Установление принципов интеграции и потребления данных.
- Разработка других руководств, политик и структур, например, для выборочного аудита журналов SQL-запросов или работы с внешними данными.

Я рассмотрю некоторые из них более подробно позже в этой главе. Чтобы поддерживать все эти действия и контроль над ними, группе управления данными нужно соответствующее оборудование, в частности инструменты и репозитории метаданных.

Люди: доверительные и этические, социальные и экономические соображения

Орган управления данными играет важную роль в развитии культуры ответственного использования данных. Он должен направлять организацию при проведении круглых столов по этике данных, при принятии решений и внедрении правил и принципов. Эти форумы должны быть репрезентативными и могут включать работу с клиентами. Цель — повысить осведомленность, установить принципы и развить культуру с этическим мышлением и четкой подотчетностью.

Орган управления данными должен установить процессы, обеспечивающие сбор информации о том, как они используются, откуда возникли, какие соображения были приняты во внимание и какие этические дилеммы обсуждались. Это может привести к появлению форм с общими рекомендациями, которые люди должны заполнить, прежде чем можно будет использовать данные.

Технологии: золотой источник, владение приложениями и их администрирование

Рассмотрим технологический аспект, функции приложений и метаданные, которые с ними связаны. Как мы уже знаем, важно обеспечить прозрачность и надежность данных. Для построения доверия нужно, чтобы группа управления данными и другие группы сделали метаданные (данные о данных) доступными централизованно с помощью инструментов и платформ. Сложность здесь в том, что метаданные часто разрознены: они доступны только в контексте платформы или инструментов¹. В связи с этим необходима всеобъемлющая стратегия для более точной идентификации и классификации приложений, данных и их владельцев. Кроме того, нужна архитектура с поддерживающими приложениями и репозиториями. Чтобы управление данными было успешным, оно должно поддерживаться следующими возможностями.

- *Репозиторий приложений.* Это автономное центральное хранилище или система управления ИТ-сервисами (<https://oreil.ly/sIKV->), которая отслеживает все уникальные приложения и их владельцев. Кроме того, в этом репозитории может храниться информация о состоянии производства, продуктах поставщиков, а также о рейтингах конфиденциальности, целостности и доступности приложения на основе имеющихся элементов управления. Для организации такого репозитория можно использовать сторонний программный компонент, например ServiceNow (<https://www.servicenow.com/>).
- *Список золотых источников (List of Golden Sources, LoGS) и наборов данных.* Это автономный репозиторий, в котором хранится общая информация обо всех уникальных данных, включая их принадлежность, по всей организации. Данные, как описано в главе 6, регистрируются на логическом уровне наборов данных, потому что у одного приложения может быть несколько владельцев данных, а несколько физических наборов данных могут происходить из одного приложения. Поэтому отношения между наборами данных и приложениями являются отношениями типа «многие ко многим».

¹ Большая часть главы 10 посвящена решению проблем распределения метаданных.

Наконец, приложение LoGS может отслеживать классификации отдельных элементов данных, таких как личные данные, цели использования, доступ к корпоративным данным и ограничения.

- *Администрирование соглашения о совместном использовании данных (data sharing agreement administration, DSAA).* Это автономное приложение, которое регистрирует все соглашения о совместном использовании данных между владельцами, пользователями и потребителями, включая цель обмена данными, меры предосторожности, условия изменения, гарантии, ограничения конфиденциальности, дату действия и т. д. DSAA играет важную роль в архитектуре безопасности, которая будет обсуждаться во второй части этой главы. DSAA, как вы узнаете позже, действует как точка администрирования политики (Policy Administration Point, PAP) или точка получения информации о политике (Policy Information Point, PIP) для предоставления информации с целью оценки и принятия решений об авторизации. Эти термины объясняются в следующем разделе — «Безопасность данных».
- *Репозиторий метаданных.* Репозиторий метаданных — это БД и связанные с ней инструменты, которые помогают пользователям находить информацию о данных и управлять ею. Вы можете использовать и сторонний программный компонент, например Alation (<https://www.alation.com/>) или Lumada (<https://oreil.ly/GUQNZ>), в качестве репозитория метаданных. Современные репозитории обычно могут также хранить дополнительные метаданные, такие как информация о владельцах, происхождении, источниках, метках, классификациях и т. д.
- *Сканеры данных и инструменты профилирования.* Сканеры и инструменты профилирования важны при обработке и изучении данных для сбора такой информации, как уровень качества данных, информация о схеме, а также о преобразовании и происхождении. Более продвинутые каталоги данных предлагают эти функции по умолчанию, но есть также предложения, предназначенные для сканирования и профилирования. Tamr (<https://www.tamr.com/>), решение для управления основными данными, оптимизировано для профилирования с использованием машинного обучения и обратной связи от человека.

Все эти инструменты, связанные с управлением данными, должны работать вместе, чтобы обеспечить интегрированное представление всех данных. Вы можете сделать это с помощью коммерческих инструментов или самостоятельно разработать модели метаданных. Кстати, это не означает, что нельзя передать отдельные детали инструментам, оптимизированным для определенных задач.

Данные: золотые источники, золотые наборы данных и классификации

Чтобы лучше объяснить принципы управления данными, я буду использовать метаданные (данные для управления данными) для построения модели, которую мы обсуждали в главе 6. В этой главе вы узнали о владельцах данных, золотых источниках, золотых наборах данных и золотых элементах данных, которые можно связать вместе. Для классификации и построения соглашений о совместном использовании данных я буду продолжать применять ту же модель, в которой объединяются управление, физические данные и соглашения. Начнем с обновленного изображения модели и рассмотрим изменения.

Уровень управления данными

Чтобы связать данные с правом собственности на них, мы используем *золотые наборы данных*: независимые от технологий представления, применяемые для классификации и для элементов данных с одинаковыми характеристиками — владелец, созданное приложение, классификации безопасности, поведение, контекст и др. Золотые наборы данных и их элементы важны, ведь они являются связующим звеном между бизнес-объектами, физическими данными, интерфейсами и моделями данных. Каждый из них принадлежит одному уникальному владельцу области. Это очень важно, потому что разные области иногда пытаются заявить о праве собственности на физические наборы и модели данных.

На рис. 7.2 вы можете заметить, что золотые наборы данных связаны как с владельцами, так и с элементами данных. Владельцы данных, выделенные фиолетовым цветом, — представители бизнеса в областях. Они обычно являются наблюдателями за данными или владельцами продуктов и несут ответственность за данные. Остальные объекты, связанные с владением набором данных, показаны зеленым. Эта информация нужна для сохранения прав собственности на золотые источники, уникальные наборы и элементы данных.

Приложения считаются золотыми источниками, если они распространяют данные. На их основе создаются наборы данных. Каждый из них содержит несколько (*золотых*) элементов данных: элементарные единицы информации, которые имеют точное значение или точную семантику, о которых вы узнали в главе 2.

К зеленому набору объектов были добавлены классификации данных: категории, в которых данные упорядочены по релевантности, чтобы их можно было использовать и защищать более эффективно. Для безопасности эти классификации данных могут перезаписывать или расширять любые отдельные элементы данных. Например, признак информации, позволяющей установить личность

(personally identifiable information, PII), может быть установлен для отдельных элементов данных, а также унаследован через одну из классификаций. Мы вернемся к этому аспекту в следующем пункте.

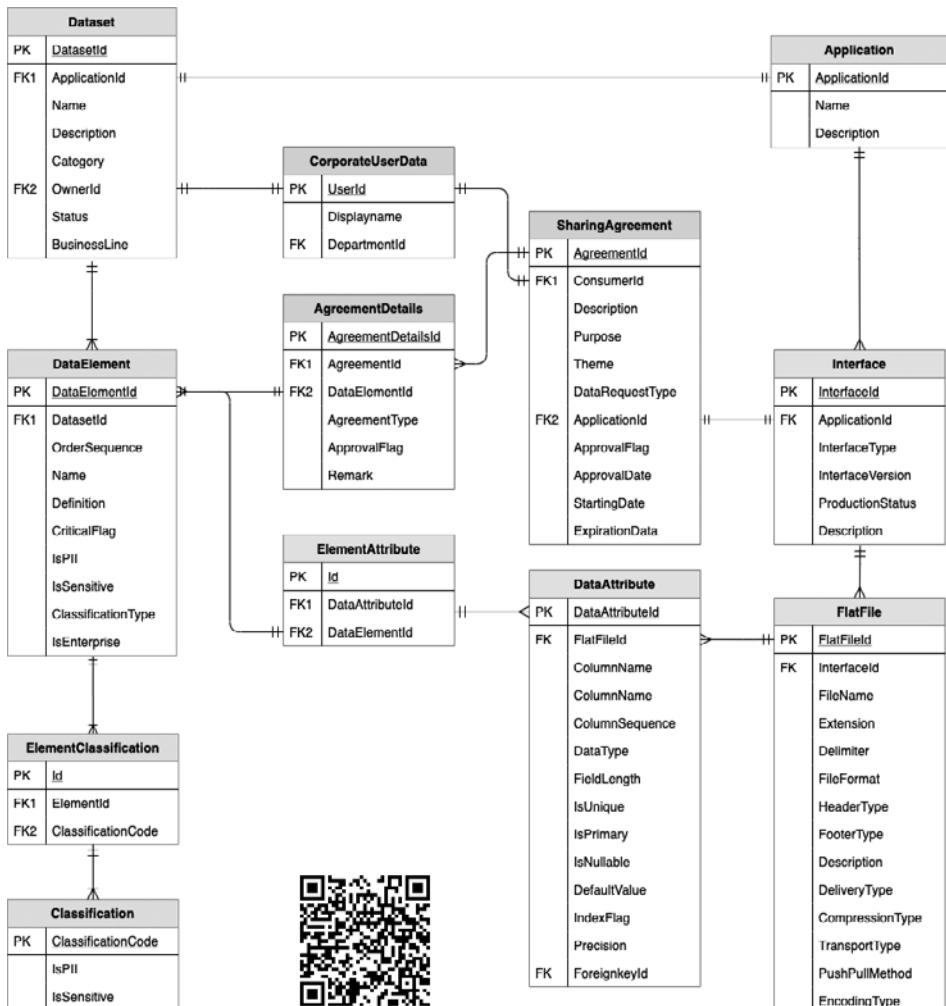


Рис. 7.2. Эталонная модель, объединяющая владение данными, физическое представление данных и соглашения

Объекты, показанные синим цветом, представляют *физические объекты данных*. Эти метаданные, как описано в главе 6, обеспечивают физическое представление данных, которое используется для чтения, поиска или записи. Такая информация может быть настолько подробной, насколько это необходимо,

с интерфейсами, атрибутами и т. д. Она основана на существующих приложениях и может быть связана с элементами данных через промежуточную таблицу `ElementAttribute`.

В середине, выделенное фиолетовым цветом, находится DSAA. Соглашения о совместном использовании данных (DSAA), как вы узнали из главы 2, представляют собой формальные контракты, которые четко документируют, какие наборы и элементы данных используются совместно несколькими областями и как именно. Они собирают информацию об объеме, конфиденциальности, ограничениях по назначению и дополнительных нюансах.

Владельцы данных и пользователи, заключая соглашения об обмене данными, должны тщательно сохранять всю информацию, связанную с потреблением, чтобы упростить мониторинг использования данных. Поскольку пользователи могут работать с несколькими наборами данных одновременно, важна дополнительная таблица: `AgreementDetails`. Она позволяет связать несколько элементов данных от разных владельцев с одним соглашением. Кроме того, она позволяет зашифровывать, замаскировать или исключать отдельные элементы в соглашении.



Если вы не создаете DSAA, а напрямую используете приложение безопасности, вы будете тесно связаны с конкретным поставщиком. Многие приложения безопасности работают только изолированно или поддерживают несколько технологий. Если вам нужен целостный, независимый взгляд на безопасность всех данных, необходимо разделить их!

Все метаданные, хранящиеся в этой модели, должны быть доступны для обнаружения, чтобы использование стало прозрачным для организации. Я рекомендую сделать репозитории и приложения открытыми и прозрачными. Их расширение с помощью дополнительных API делает данные доступными для поиска и позволяет использовать их в других проектах или процессах. Вариант соглашения о совместном использовании данных представлен в примере 7.1.

Пример 7.1. Образец соглашения о совместном использовании данных

```
{
  "@AgreementId": "1342",
  "contactPoint": {
    "ConsumerId": "894",
    "fn": "Marketing Department",
    "hasEmail": "mailto:email@example.com"
  },
  "ApplicationId": "201",
  "Description": "<p>Это соглашение о совместном использовании данных определяет порядок использования маркетинговых данных в сфере управления персоналом. Эти данные могут использоваться только отделом кадров, потому что они могут содержать конфиденциальную информацию о сотрудниках и клиентах.</p>",
}
```

```
"DataRequestType": "dcat:FlatFile",
"IssuedDate": "2019-11-26",
"StartingDate": "2019-12-01",
"ExpirationDate": null,
"theme": [
    "AD_HOC_USAGE"
],
"purpose": [
    "HR_ONLY"
],
"dataElements": [
    "1256", "1257", "1258", "1259", "1260", "2341", "3678", "7864"
],
"ApprovalDate": "2019-11-27",
"ApprovalFlag": true
}
```

Для классификации и описания наборов данных нужны метаданные. Сложность здесь в том, что иногда в одном приложении хранятся данные, принадлежащие разным владельцам. Это связано с тем, что некоторые приложения могут быть реализованы как общие бизнес-возможности, используемые сразу несколькими областями. Поэтому данные могут принадлежать разным областям и иметь нескольких владельцев.

Чтобы привязать детализированные физические данные к разным владельцам, нужно использовать один из этих двух подходов.

- *Физическая группировка данных.* Этот метод назначения прав собственности работает за счет сохранения данных в разных функциональных разделах и применения к ним различных мер безопасности. Данные, принадлежащие разным владельцам, должны быть изолированы. Например, если приложение выдает данные, принадлежащие двум разным владельцам, то они должны быть разделены на два отдельных файла. На рис. 7.3 показано, как это работает.

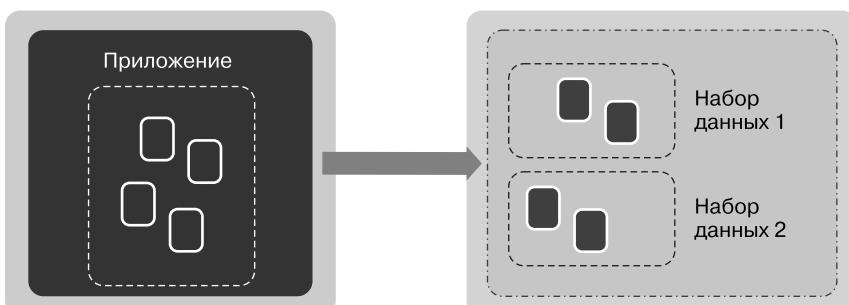


Рис. 7.3. Для правильной классификации и описания данные нужно разделить на отдельные наборы

Чтобы привязать к элементам данных физические атрибуты, которые описывают их владение, используется промежуточная таблица `ElementAttribute` (см. рис. 7.2). Каждый физический объект данных в разделе или папке будет связан с отдельным элементом данных с другим владельцем.

- *Группировка путем инкапсуляции метаданных внутри записей данных.* Для этого метода нужно встроить метаданные в данные, например, дополнив каждую запись дополнительным кодом принадлежности или классификации набора данных. Если все сделано точно, правила доступа к данным можно распространить на блоки (динамически определяемые фрагменты или определенные части данных), столбцы, записи или даже их комбинации. Разберем этот аспект подробнее.

Например, чтобы обеспечить детализированный доступ к данным в RDS, можно объединить метаданные из DSAA, списка администрирования золотых источников, метаданных схемы и самих данных.

Для метаданных, связанных с данными или встроенных в них, я использую в этом примере файл CSV. На рис. 7.4 показан файл, который может храниться в RDS. Здесь есть на что обратить внимание. Во-первых, внутри данных есть элемент метаданных `DatasetId` ①. Он соответствует уникальному идентификационному номеру золотого набора данных, который зарегистрирован в списке администрирования золотых источников. Во-вторых, имеются дополнительные атрибуты метаданных ② для расширенной фильтрации. Они могут использоваться для фильтрации определенных строк или предоставления доступа к определенному фрагменту данных. В-третьих, имеются метки конфиденциальности. Определив их должным образом, можно разрешить доступ только к нечувствительным данным или отфильтровать конфиденциальные сведения. Например, для потребителя можно создать виртуальное представление.

PK	1 DatasetId	ContactName	CompanyName	2 Division	ProspectId	Country	3 Sensitive
...
4524	001	Jon Snow	Braavos N.V.	A	A001	NL	Y
4525	001	Arya Stark	Braavos N.V.	A	A001	NL	Y
4526	001	Gregor Clegane	Qarth	B	A001	UK	Y
4527	002	Tormund Giantsbane	Meereen	A	Q001	NL	Y
...

Рис. 7.4. Предоставляя данные и метаданные, можно обеспечить детализированный доступ к данным

Представляемые вместе с данными атрибуты метаданных играют важную роль в соглашениях о совместном использовании. Например, соглашение может предусматривать возможность доступа к столбцу **Country** со значением **NL** для определенного пользователя, только если столбец **DatasetId** содержит значение **001**. Соглашение о совместном использовании данных, содержащее всю эту информацию, является основой для поддержки расширенных политик безопасности.

Метки данных и классификации

Связь между управлением данными и безопасностью становится очевиднее при маркировке и классификации данных. Сначала вы определяете классификации, а затем используете метки для маркировки данных, указывая, к каким классам относятся те или иные данные.

Метки и классификации — важные аспекты управления данными. Они применяются для группировки данных по соответствующим категориям, чтобы их можно было использовать и защищать более эффективно. На базовом уровне процесс классификации упрощает поиск и извлечение данных, но на более продвинутом он может использоваться для обеспечения безопасности, соответствия и регулирования. Классификация безопасности, например, может действовать как ограничение на доступность данных для пользователей. Список классификаций данных лучше всего вести централизованно, и он не должен быть слишком длинным.



Нужно ли классифицировать неструктурированные данные, такие как документы, изображения и файлы журналов? Да, не следует классифицировать только структурированные данные. Неструктурированные и полуструктурные данные можно идеально классифицировать, используя одни и те же схемы.

Хорошие примеры меток безопасности: Идентификационные данные клиента, Номер социального страхования, Религия, Пол, Информация о сотруднике, Коммерческая информация о компании и Внешняя информация. Следующий шаг — привязка этих меток к классификациям безопасности. Хорошими примерами в этом случае могут служить метки Публичное, Частное и Ограниченнное. Метки для информации, раскрытие которой может нанести вред отдельным лицам, обычно связываются с классом Ограниченнное. Другие личные или конфиденциальные данные, такие как личные имена и номера социального страхования, являются хорошими кандидатами на отнесение к классу Частное. Все остальные элементы можно отнести к категории Публичное. При необходимости можно расширить список дополнительными атрибутами, такими как Конфиденциальное для данных сотрудников или информации об организации.

ЗАЩИТА ПЕРСОНАЛЬНЫХ ДАННЫХ И СОБЛЮДЕНИЕ НОРМАТИВНЫХ ТРЕБОВАНИЙ

Ключом к успешному управлению личными данными и согласию пользователей является сочетание управления данными, безопасности и управления основными данными. Управление данными сосредоточено на управлении процессами, классификациями и метками. Это позволяет увидеть, какие данные конфиденциальны и требуют особого внимания. Безопасность данных основана на ограничении доступа к данным. Управление основными данными, как вы узнаете из главы 9, сосредоточено на правилах сопоставления и идентификации данных. Благодаря этой дисциплине можно однозначно идентифицировать клиентов и знать, какой тип согласия был дан. Таким образом, соблюдение нормативных требований, таких как GDPR и CCPA, не означает сосредоточение внимания на какой-то одной области управления данными. Путь к соблюдению требований начинается с идеи комплексного интегрированного управления данными.

Преимущество использования меток и значков в том, что они не связывают правила доступа с данными и метаданными. Например, если правила вынуждают вас считать коммерческую информацию конфиденциальной, не нужно спешить выполнять маркировку всех данных. При хорошо составленной модели метаданных ограничения должны наследоваться автоматически.

Классификации важны, потому что в политиках управления они автоматически ограничивают доступ к данным. Это особенно важно для критических данных и данных, раскрытие которых несет высокие риски. Когда данные доставляются или отображаются через уровень данных, каждый элемент данных должен быть классифицирован¹.

Классификация целей

Чтобы лучше настроить использование и обнаружение данных, попробуйте использовать классификации целей для ограничения использования данных конкретными сценариями или вариантами использования. Этот список лучше всего вести централизованно, и его не следует «засекречивать». Если он будет слишком длинным, пользователям будет сложно сделать выбор. Рассмотрим примеры классификаций целей.

- НАУКА_О_ДАННЫХ — для исследования и анализа данных.
- ОСОБЫЙ_СЛУЧАЙ — для неповторяющейся работы, например создания отчетов на лету.
- УПРАВЛЕНИЕ_ПЕРСОНАЛОМ — для использования в отделе кадров.

¹ При классификации физических данных, например строк, классификации или ссылки на классификации инкапсулируются в самих данных.

- **ФИНАНСОВАЯ_ОТЧЕТНОСТЬ** — для составления финансовой отчетности и отчетов о рисках.
- **КАЧЕСТВО_ДАННЫХ** — для исследования и определения качества данных.

Классификация целей влияет на то, что пользователи могут видеть и делать с данными. Их можно комбинировать с классификациями данных. Так, **УПРАВЛЕНИЕ_ПЕРСОНАЛОМ** можно объединить с меткой Конфиденциально, чтобы гарантировать доступность конфиденциальных данных только для сотрудников отдела кадров.

Идентификация схемы

Чтобы лучше классифицировать и идентифицировать данные, важно полагаться на метаданные, предоставляемые при получении или передаче информации через уровень данных. В примере 3.1 вы видели один из вариантов метаданных, описывающих схему XML, которые могут доставляться вместе с данными в хранилище только для чтения. Вместо доставки метаданных схемы вместе с данными также можно попросить владельцев приложений изменить или реализовать поддержку схемы (метаданных) с портала. Соединение данных физической схемы и бизнес-метаданных важно для надежного управления данными, включая их безопасность.

Поддержание соглашений о совместном использовании данных, списка приложений (золотых источников), уникальных золотых наборов данных и соответствующих приложений, владельцев данных и физических отношений — жизненно важная деятельность по управлению данными. У каждого приложения должен быть один владелец, и оно потенциально может использовать нескольких золотых наборов данных. Каждый золотой набор данных связан с одним приложением и одним назначенным владельцем данных. Для проверки общей целостности этих метаданных управление данными должно использовать те же инструменты качества данных, которые применяются для профилирования любых других (предметных) данных.

Безопасность данных

Безопасность данных, как описано в главе 1, охватывает все, что связано с защитой данных: людей, процессы и технологии. Очень важно не допускать попадания в руки преступников, хакеров, злоумышленников и конкурентов персональных данных, информации о личном здоровье, о продажах и интеллектуальной собственности.

Проблема в том, что данные непостоянны и больше не хранятся в едином монолите. Они децентрализованы: распределены по множеству систем и сред. Распределенные данные усложняют контроль за действиями пользователей после получения

данных. Например, будут ли они комбинировать данные и проводить неэтичный анализ? Сложно защитить данные, не зная их полного контекста. Данные отдела кадров, связанные с реорганизацией имеют другие значение и ценность, чем данные, связанные с организацией пикника или другого праздничного мероприятия.

Более серьезная проблема — большой объем данных, с которым крупные компании имеют дело. Уровень данных — это плавильный котел: в нем объединяются источники данных, потребности и запросы владельцев и потребителей. Каждый отдельный источник и соответствующие ему элементы данных имеют разный контекст, качество, классификацию, привязку к цели, владельца, рейтинг и т. д. По мере того как новые источники с большим количеством данных и цифровых идентификаторов вводятся на уровень данных, их сложность значительно возрастает. Описание, управление и защита всех этих данных требует огромного множества правил и политик безопасности.

Безопасность данных в крупных организациях обычно обеспечивается специальным отделом безопасности, возглавляемым руководителем по информационной безопасности (chief information security officer, CISO). Руководитель по информационной безопасности — руководитель высшего звена, отвечающий за формирование и поддержание идеи, стратегии и процессов предприятия, обеспечивающих надлежащую защиту информационных активов и технологий. Он работает в тесном сотрудничестве с отделами архитектуры предприятия и управления данными, чтобы гарантировать правильное внедрение видения и целей безопасности в общую организацию предприятия.

Текущий разрозненный подход

Существует огромный разрыв между тем, что внедрило большинство предприятий, и тем, что отстаивают эксперты. Многие консалтинговые компании используют термин «информационно-центрическая безопасность», чтобы подчеркнуть, что безопасность должна быть сосредоточена на самих данных, а не на сетях, серверах или приложениях. На практике текущие реализации безопасности сосредоточены в основном на разрозненных хранилищах и озерах данных, поскольку большие объемы данных и их комбинации могут означать высокие риски и повышенное внимание.



Классическая комбинация — Apache Atlas (<https://oreil.ly/jQVUF>) и Apache Ranger (<https://oreil.ly/8UCYR>). Apache Atlas используется для описания, классификации и определения политик данных. Apache Ranger — это механизм авторизации, с которым работают другие компоненты экосистемы Hadoop, чтобы обеспечить безопасный доступ к данным. Хотя такая комбинация логична, архитектура корпоративных данных всегда имеет более широкий охват, чем просто Hadoop. Поэтому такие компоненты должны интегрироваться с корпоративными репозиториями метаданных.

Для API и событийно-ориентированных архитектур модель безопасности во многих компаниях реализуется при помощи различных процедур, классификаций, инструментов и возможностей. Безопасность в этих архитектурах обычно ориентирована на конечные точки, а не на данные, проходящие через эти платформы. Классификация данных и окружающий контекст практически не играют роли в сценариях эксплуатации. На мой взгляд, модель безопасности, ориентированная на данные, должна фокусироваться на всех данных и шаблонах распределения. Это именно то, что мы будем изучать дальше.

Единая защита данных для архитектур

Рекомендуемый подход к реализации безопасности данных — с самого начала включить требования безопасности в архитектуру. Для успешного устранения рисков безопасности необходимо сосредоточить внимание на двух уровнях контроля. Первый уровень — это *средства управления безопасностью данных*, включая управление доступом, маскировкой, шифрованием и мониторингом использования данных, а также поставщиками идентификационной информации. Второй уровень — уровень инфраструктуры — ориентирован на изоляцию, сетевое шифрование, использование межсетевых экранов и т. д.

Управление доступом на основе ролей и атрибутов

Основная задача управления данными — организация доступа к данным, то есть обеспечение возможности чтения или извлечения данных, хранящихся в базе данных или приложении. Существует две популярные модели управления доступом к данным: *управление доступом на основе ролей* (role-based access control, RBAC) и *управление доступом на основе атрибутов* (attribute-based access control, ABAC). Рассмотрим эти модели и определим, как их использовать.

Большинство компаний начинают с RBAC, метода безопасного доступа, основанного на определении ролей и соответствующих привилегий. Идея этой модели в том, что каждому пользователю (сотруднику) назначается роль с набором разрешений и ограничений. Пользователь может получить доступ к данным и выполнять операции, только если назначенная роль имеет соответствующие разрешения.

Главный недостаток RBAC — «взрывной рост количества ролей». В любой крупной организации выявится большое количество разных ролей. Департаменты, подотделы и многие функции имеют тонкие различия. Управление всеми этими ролями может стать затруднительным. Если вы не хотите использовать грубую ролевую модель, то, скорее всего, получите тысячи разных ролей. Другая проблема с RBAC в том, что эта модель статична и не принимает во внимание

контекстную информацию — местоположение пользователя, время и информацию об устройстве.

ABAC — более продвинутый метод безопасности, устраняющий недостатки RBAC. Его политики основаны на различных атрибутах данных и могут исходить из самих данных, таких как классификации и свойства метаданных, или из системного контекста и пользователей (роли, географическое положение, свойства устройства и т. д.) и действий, которые должны быть выполнены с данными (чтение, вставка, обновление или удаление). ABAC сопровождается рекомендуемой архитектурой, показанной на рис. 7.5, которая включает следующие четыре компонента.

- *Точка применения политики (Policy Enforcement Point, PEP)*. Компонент PEP несет ответственность за защиту данных. Он проверяет запрос данных и генерирует запрос авторизации, который отправляется в точку принятия решения о политике для проверки и утверждения.
- *Точка принятия решения (Policy Decision Point, PDP)*. PDP — это ключевой компонент ABAC. Он проверяет входящие запросы на соответствие политикам безопасности, определенным для всех атрибутов. PDP возвращает решение об утверждении или отклонении на основе результатов политики. Затем он сообщает PEP о своем решении и причине утверждения или отклонения.

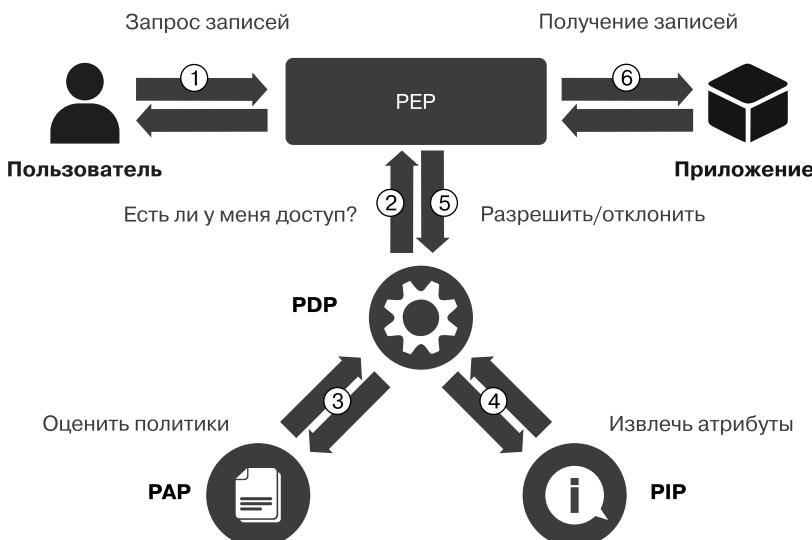


Рис. 7.5. Модель ABAC имеет стандартную эталонную архитектуру

- *Точка получения информации о политике (Policy Information Point, PIP)*. Компонент PIP позволяет PDP использовать данные внешнего источника, такие как атрибуты пользователя, вызывая поставщика идентификации.
- *Точка администрирования политики (Policy Administration Point, PAP)*. PAP – это репозиторий, который управляет всеми политиками безопасности предприятия. Типичный компонент PAP также обеспечивает мониторинг и журналирование и имеет удобный интерфейс.

Конкретный пример ABAC – конфиденциальные данные отдела кадров, отмеченные атрибутом метаданных: *информация о сотруднике*. Если человек, не связанный с персоналом, пытается получить доступ к данным, отмеченным этим тегом, то PDP отклоняет запрос. Атрибуты в ABAC также можно комбинировать. Пример с персоналом также можно распространить на среду, из которой человек пытается получить доступ к данным: работа из дома или из корпоративной среды может привести к разным ответам.

Поставщики удостоверений

Когда приложения работают в объединенной или распределенной сети, в моделях RBAC и ABAC появляются важные компоненты: *поставщики удостоверений* (identity provider, IDP). IDP – это внешние источники для получения атрибутов от пользователя. Они предлагают аутентификацию пользователя как услугу и часто используются для обработки процессов входа в другие системы – веб-сайты, приложения CRM и файловые серверы. Наиболее известные реализации IDP основаны либо на Microsoft Active Directory (AD) (<https://oreil.ly/qyzBv>), либо на OpenLDAP (<https://www.openldap.org/>).

На данный момент есть два популярных открытых протокола для аутентификации: SAML и OpenID Connect. SAML (Security Assertion Markup Language – язык разметки утверждения безопасности) базируется на стандарте XML для обмена данными авторизации и аутентификации. Он основан на продукте Технического комитета служб безопасности OASIS. OpenID работает так же, как SAML, но он более открытый и используется такими крупными компаниями, как Microsoft, Facebook, Google, PayPal и Yahoo. В этом контексте иногда также упоминается OAuth. В отличие от SAML и OpenID, OAuth используется только для авторизации, но не для аутентификации. Еще одна концепция, о которой вы можете услышать, – *технология единого входа* (single sign-on, SSO). При использовании SSO пользователи должны пройти аутентификацию только один раз (представив свою смарт-карту (<https://oreil.ly/ToNkQ>) или учетные данные, такие как имя пользователя и пароль) и получить доступ к нескольким приложениям без повторной аутентификации.

Эталонная архитектура безопасности и подход к контексту данных

Теперь рассмотрим два архитектурных блока для защиты уровня данных и решения проблемы экспоненциально растущего объема данных, их атрибутов и правил безопасности, связанных с ними. Для решения этой проблемы управления сложностью используются два основных архитектурных блока: *механизм политики безопасности* (Security Policy Engine, SPE) и *интеллектуальный механизм обучения* (Intelligent Learning Engine, ILE)¹. В этом и следующих разделах я объясню, что это за компоненты и как они работают с существующими строительными блоками архитектуры.

Эталонная архитектура безопасности основана на модели безопасности ABAC, потому что ABAC, в отличие от RBAC, предлагает более детализированные правила безопасности, которые нужны для защиты комбинации разных атрибутов (данных). Как вы понимаете, атрибуты будут предоставляться из окружающих репозиториев метаданных, которые мы обсуждали в разделах управления данными: репозитория приложений, LoGS, классификации данных и классификации целей. Все они тесно связаны с обеспечением безопасности данных.

Начнем с иллюстрации эталонной архитектуры безопасности (рис. 7.6), а затем рассмотрим компоненты и последовательность операций.

Поставщики и потребители данных, а также уровни уже вам знакомы, но четыре компонента безопасности на вершине архитектуры упоминаются здесь впервые. Каждый компонент безопасности играет важную роль и должен работать в тесном контакте с другими. Все они пронумерованы на иллюстрации.

1. Механизм SPE – это точка принятия решений (PDP). Он отвечает за управление закрытой системой элементов (данные, поставщики, потребители, отношения и т. д.) и их атрибутов и работает с ограниченным числом переменных атрибутов, поэтому его можно преобразовать в политики и правила управления безопасным доступом к данным.
2. ILE предоставляет SPE возможности обучения. Он отвечает за управление системой в ее развивающемся состоянии и изучение моделей, которые SPE может использовать. Этот компонент по определению не относится к ста-

¹ Архитектура безопасности расширяема: вы можете подключать другие инструменты и платформы для обеспечения безопасности. Мы обсудим это в разделе «Практическое руководство» на с. 250.

тической закрытой системе правил. Он наблюдает за динамикой открытой системы, выявляет кластерные закономерности принципов и атрибуты безопасности и в дальнейшем выступает в качестве управляющего механизма и как расширение правил SPE и политик.

3. Уровень общих компонентов — это абстрактный уровень сервисов, который позволяет реализовать функции безопасности, такие как PEP, базу данных журналов и механизм предоставления инфраструктуры.
4. Хранилища метаданных содержат все атрибуты данных (включая атрибуты безопасности и конфиденциальности), а также данные, поступающие напрямую через SPE, и улучшения, внесенные вручную сотрудником службы безопасности. Объединенные хранилища метаданных действуют как PIP для PAP и SPE.

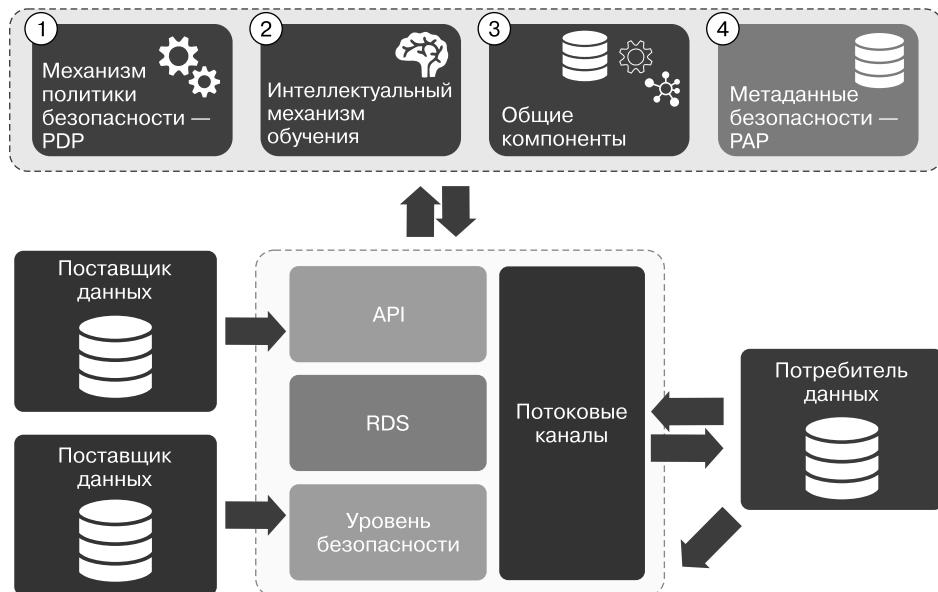


Рис. 7.6. Эталонная архитектура безопасности включает четыре компонента безопасности: SPE, ILE, общие компоненты и хранилище метаданных

Компоненты безопасности тесно взаимодействуют друг с другом, поэтому дальше я объясню процесс интеграции (предоставления) и потребления данных. После каждого шага я буду ссылаться на компоненты и объяснять их роль в архитектуре. К концу следующего подраздела вы будете понимать, как эти компоненты работают вместе.

Процесс безопасности

Безопасность на уровне данных обеспечивается отслеживанием перемещения данных через одну из архитектур интеграции. Чтобы объяснить, как это происходит, давайте поработаем с эталонной моделью (рис. 7.7), основанной на архитектуре безопасности, описанной в предыдущем разделе.

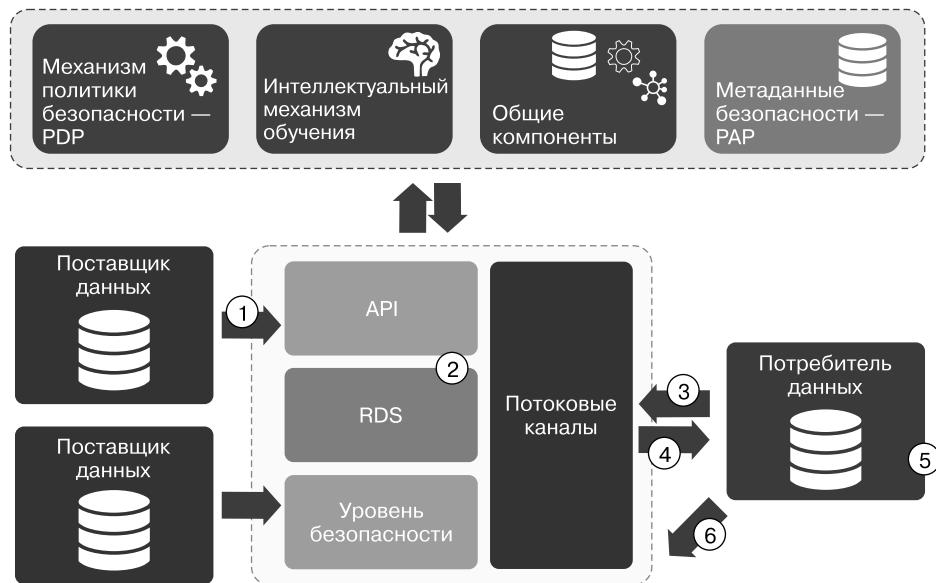


Рис. 7.7. Эталонная архитектура безопасности

Процесс приема, проверки и использования данных состоит из нескольких этапов. Каждый помечен цифрой и будет более подробно описан в следующих подразделах.

Прием данных

Первый шаг ❶ приема данных или их экспорт через уровень данных начинается с предоставления самих данных и дополнительной информации в виде метаданных. На этом этапе мы ожидаем, что каждый уникальный набор данных будет зарегистрирован в LoGS вместе с соответствующим владельцем. Предварительное условие для этого шага — приложение уже должно быть зарегистрировано в репозитории. Кроме того, мы ожидаем, что все бизнес-метаданные будут связаны с физическими данными. Во время процесса интеграции данные и их цели должны быть помечены и классифицированы.

Эти процессы можно ускорить за счет поддержки возможностей машинного обучения для автоматической классификации данных. Подобные программы для профилирования могут автоматически определять номера социального страхования и кредитных карт, имена и адреса клиентов. По мере предоставления большего количества метаданных модели искусственного интеллекта и машинного обучения становятся умнее, ускоряют процесс, предоставляя предложения, и повышают надежность классификации.

Важно сохранить контекст безопасности для оперативных сценариев использования, ведь эти данные могут повлиять на оперативную аналитику. Это означает, что данные локальной области, такие как метки конфиденциальности, страны, диапазоны областей и т. д., должны быть частью данных или должны предоставляться во время процесса интеграции. Без привязки этого контекста к данным потом будет сложно реализовать более детальные правила безопасности.

Данные в состоянии покоя

Из-за того что данные находятся в состоянии покоя ❷, вступают в силу дополнительные меры безопасности. В зависимости от склонности компании к риску и политик безопасности данные могут быть автоматически токенизированы и зашифрованы (к примеру). Этот шаг применяется только к RDS и потоковой архитектуре, потому что в этих архитектурах данные могут сохраняться достаточно долго. Передаваемые данные должны быть зашифрованы на уровне инфраструктуры с помощью других протоколов, таких как Transport Layer Security (TLS) или Secure Sockets Layer (SSL) (<https://oreil.ly/J1T1b>).

Доступ к данным

Доступ к данным ❸ для пользователей немного сложнее, потому что он включает несколько дополнительных шагов. Потребление данных начинается с соглашения о совместном использовании, в котором и владелец (-цы), и пользователь данных соглашаются с вариантами потребления и целями использования данных. Без этого соглашения нельзя разрешать потребление данных.



Соглашения о совместном использовании данных применяются только тогда, когда у пользователей еще нет доступа к данным. Если пользователям уже разрешено просматривать данные для определенных целей, доступ должен быть предоставлен автоматически и нет необходимости соблюдать процесс соглашения.

После согласования контракт сохраняется в DSAA. Этот контракт и соответствующие ему атрибуты — важные элементы для SPE, которые мы обсудим

далее. Если у вас есть соглашение о совместном использовании, можно начать потребление данных. Именно здесь происходит волшебство, когда несколько компонентов объединяются и работают вместе, чтобы проверить, следует ли предоставлять доступ. Этот процесс и соответствующие задачи являются основой для любых дополнительных решений, связанных с доступом к данным, если тот разрешен.

1. Собираются атрибуты из PIP. Они предоставляют информацию о пользователях или приложениях, их отделах, группах, ролях, целях, дате начала работы в организации, дате последнего успешного входа в систему и т. д. Вся эта информация будет собрана и объединена для следующей серии задач.
2. Следующая задача — собрать метаданные: классификации золотых источников, метаданные приложений (такие как конфиденциальность, целостность и рейтинги доступности), состояние производства, классификации данных, соответствующие бизнес-возможности, независимо от того, осуществляется ли доступ к данным атомарно или в сочетании с другими наборами данных и т. д.
3. Собираются метаданные целевого приложения (приложений), такие как рейтинги приложения-потребителя, состояние производства и сведения о времени выполнения.
4. Запрашиваются окружающие хранилища метаданных, такие как БД журналов с предыдущим временем доступа, данные телеметрии или информация об инфраструктуре (локальной или облачной)¹.
5. Последняя задача — объединить всю эту информацию и передать ее SPE для принятия решения.

SPE принимает решения на основе набора правил, которые на практике определяют, какие типы ответов должны давать определенные комбинации атрибутов. На основании правил запроса и сопоставления потребителям данных будет предоставлен или запрещен доступ. Правила сообщают уровням потребления, какие данные разрешено видеть потребителям и как именно. Например, некоторые данные можно скрыть, а другие — обезличить с помощью методов маскировки. Последняя подзадача — отправить запрос, соответствующую информацию и решение в базу данных журнала безопасности для будущего анализа.

¹ Телеметрия — это автоматизированный процесс связи, с помощью которого измерения и другие данные собираются в удаленных или недоступных точках и передаются на принимающее оборудование для мониторинга.

Потребление данных

На основе запроса потребителя SPE автоматически решает, к каким данным можно разрешить доступ и как они должны потребляться ④. В зависимости от правил безопасности и контекста запроса могут быть вызваны дополнительные компоненты (❸ на рис. 7.6). Посмотрим на несколько сценариев.

Предоставление доступа

- Предоставление доступа происходит в режиме реального времени. Когда пользователи с SSO или приложениями пытаются получить доступ к данным, запросы перехватываются, принимаются решения и данные возвращаются.
- В архитектуре RDS автоматически создается ориентированное на потребителя «представление», которое он может использовать. В зависимости от решения SPE определенные данные могут быть скрыты, замаскированы или зашифрованы. Ожидается, что для этого типа функциональности RDS будут работать вместе с такими инструментами защиты данных, как Apache Ranger, Apache Sentry (<https://oreil.ly/k7TcZ>), Axiomatics (<https://oreil.ly/CzNI8>), Privitar (<https://oreil.ly/A3qsH>), Protegrity (<https://oreil.ly/KdnXm>) и Secure@Source от Informatica (<https://oreil.ly/j1aeY>).
- Для определенных целей (специальная отчетность, исследование данных) доступ может быть предоставлен только к определенному инструменту или на определенный срок. Например, инструментам Wrangling можно предоставить доступ к данным в определенных ситуациях только на ограниченное время.
- Данные могут быть продублированы и предварительно обработаны специально для определенного проекта или варианта использования. На этапе обработки данные могут быть замаскированы, скрыты, отфильтрованы, токенизированы или зашифрованы. Этот сценарий хорошо работает для очень больших наборов данных.
- Для вызовов и событий API определенные блоки в сообщении JSON фильтруются или удаляются с помощью дополнительного уровня безопасности.
- Платформа потоковой передачи формирует индивидуальные темы, содержащие только сообщения для конкретного потребителя.

После того как SPE предоставил доступ к данным, путешествие продолжается на стороне потребителя, где мы видим приложения и возможности.

Обработка данных на стороне потребителя

В архитектуре RDS данные защищаются, как описано в этой главе, за счет использования метаданных из различных репозиториев. Например, облачная политика может обеспечивать шифрование данных, хранящихся на стороне потребителя, или автоматически предоставлять данным определенные ресурсы.

Повторная интеграция данных

В тот момент, когда потребитель становится поставщиком данных ❾, цикл обработки потока завершается и все шаги необходимо повторить снова.

Поначалу количество метаданных и шагов может быть огромным, поэтому я рекомендую создавать архитектуру безопасности постепенно. Начните с основ, собрав самые важные метаданные и создав ограниченный набор классификаций и несколько моделей потребления. Постепенно расширяйте архитектуру за счет более продвинутых возможностей и метаданных. В конце сделайте процесс более самообслуживаемым, чтобы его можно было расширить.

Практическое руководство

Чтобы увидеть, как связаны друг с другом процесс обеспечения безопасности, репозитории метаданных и модель безопасности, давайте подробнее рассмотрим разные типы обмена информацией внутри уровня данных. Эти практические примеры помогут вам понять, как защита данных работает в более широком масштабе и какие последствия это может иметь.

Архитектура RDS

Данные защищены в архитектуре RDS, как мы рассмотрели ранее в этой главе, за счет использования метаданных из различных репозиториев. Потребление начинается с основных компонентов: информации о собственности, классификации, отношений с физическими объектами данных и соглашений о совместном использовании. С помощью этой информации мы узнаем, какие данные пользователи могут получить.

С другой стороны, у нас есть политики безопасности. Они следят за тем, чтобы пользователи могли получить доступ только к тем данным, которые им нужны для работы и вариантов использования. Другими словами, они определяют, какие элементы управления доступом к данным нужно применить к этим данным. Такие политики принимают в качестве входных данных соглашения о со-

вместном использовании и всю другую информацию, связанную с управлением данными и метаданными, и решают, разрешено ли потребление данных.

Таким образом, соглашения о совместном использовании данных и информация из дополнительных источников считаются точкой получения информации о политике (PIP) и собираются в точку администрирования политик (PAP), где они и хранятся. В зависимости от того, какие комбинации собраны и в каком поведенческом контексте находятся потребители, выполняется окончательный вызов с использованием PDP. Поэтому доступ к данным будет обеспечиваться одним из PEP, как показано на рис. 7.8.

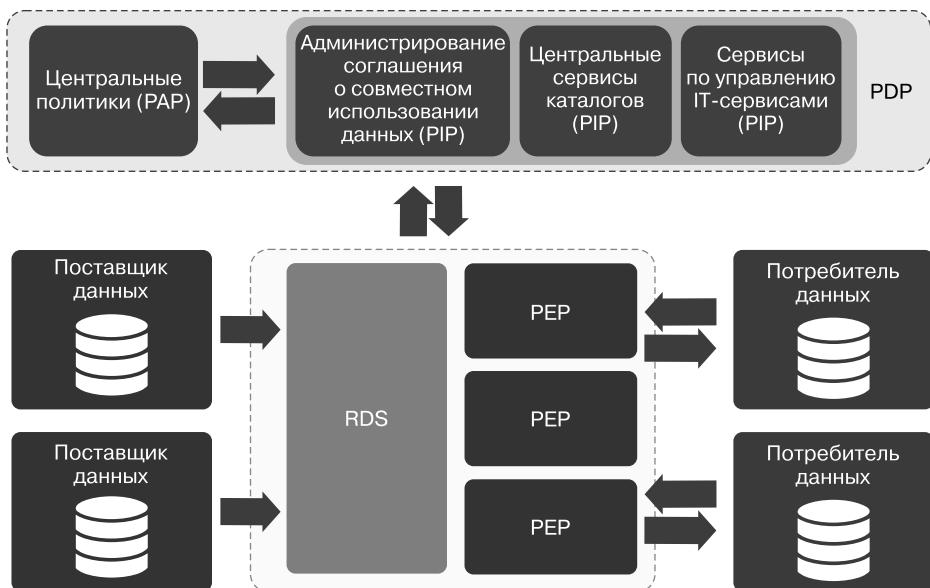


Рис. 7.8. PIP, PAP, PDP и PEP работают вместе для обеспечения безопасности в масштабах всего предприятия

Причина, по которой может существовать несколько PEP, в том, что у нас могут быть разные модели потребления и принудительного применения политик при передаче данных из RDS. У каждой модели потребления может быть собственная выделенная точка применения политик (PEP). Решения в этой схеме безопасности принимаются центральным PDP, который использует PAP и PIP в качестве входных данных.

В области решений применяются различные варианты и комбинации технологий. PAP может быть самодельным решением или предоставляться коммерческим

поставщиком. Проекты PDP и PEP тоже могут различаться. Например, если ваши RDS используют Azure SQL Server, вы можете сгенерировать операторы `CREATE`, `SECURITY`, `POLICY` для обеспечения безопасности на уровне строк (row-level security, RLS) и безопасности на уровне столбцов (column-level security, CLS) базы данных во время выполнения. Поэтому SQL Server будет перехватывать все проходящие запросы и проверять их на соответствие политикам. Если RDS используют Hadoop в качестве платформы, вы можете преобразовать политики PAP в Apache Ranger. Если вы используете сторонний инструмент, такой как Axiomatics, придется преобразовать политики в это решение. Другой вариант — дублировать и обезличивать данные для каждого нового варианта использования с такими инструментами, как Privitar. Пока вы разделяете репозиторий совместного использования данных и уровень потребления, вы получаете гибкость для использования любого нужного решения. Вы можете поддерживать различные модели потребления, последовательно применяя одну и ту же модель безопасности ко всем данным.



В этом примере столбцы метаданных кажутся фиксированными, но и здесь можно использовать метаданные, чтобы добавить динамику. Например, дополнительный файл метаданных (файл JSON) может быть доставлен с данными, описывающими схему, а также с фильтрами безопасности: классификациями или динамическими правилами, которые сообщают структуре безопасности, какой столбец искать и какие фильтры применять. Вы также можете назначать владельцам столбцы или целые наборы данных.

В этом примере мы сосредоточились на самих данных, поэтому атрибуты безопасности взяты из целевого объекта. Но мы можем использовать атрибуты запрашивающего пользователя или желаемого действия. Например, можно основывать контракты на использовании в среде специалистов по данным, а также на конкретных ролях или организационных отделах. Эти атрибуты можно сочетать и применять как выходные данные для детальной модели безопасности данных. Такая модель может быть настолько сложной, насколько это необходимо. Чем больше измерений и атрибутов вы добавите, тем прочнее будет модель.

Архитектура API

Обеспечение безопасности API в рамках SOA — сложная задача, поскольку обмен данными должен происходить в реальном времени. Безопасность обычно обеспечивается на двух уровнях: на уровне API и на уровне поставщика API.

Первый уровень безопасности обеспечивается на уровне API с использованием шлюзов API, сервисных шин предприятия и сервисных сеток. Они действуют как

серверы делегированной авторизации для надлежащей авторизации потребителей, запрашивающих ресурсы у зарегистрированных API провайдеров. В этой модели (рис. 7.9) потребители API сначала должны подтвердить свою личность ①, вызвав определенные службы аутентификации. После успешной идентификации предоставляется токен идентификации (и, не обязательно, токен обновления) ②, который позволяет потребителям API выполнять авторизованные вызовы API¹. Теперь, когда потребитель API авторизован, он может использовать этот токен для вызова других API ③. Шлюз API в этой модели перехватывает запросы и проверяет объем доступа и действительность токена. Он может напрямую запросить сервер авторизации, чтобы проверить токен. Политики безопасности API-архитектуры должны исходить из соглашения о совместном использовании данных для приложения. Либо они извлекаются из этого приложения и преобразуются в политики безопасности API, либо приложение соглашения о совместном использовании данных действует как конечная точка принятия решения по политике API. В этой модели API-шлюз фактически является вашим PEP.

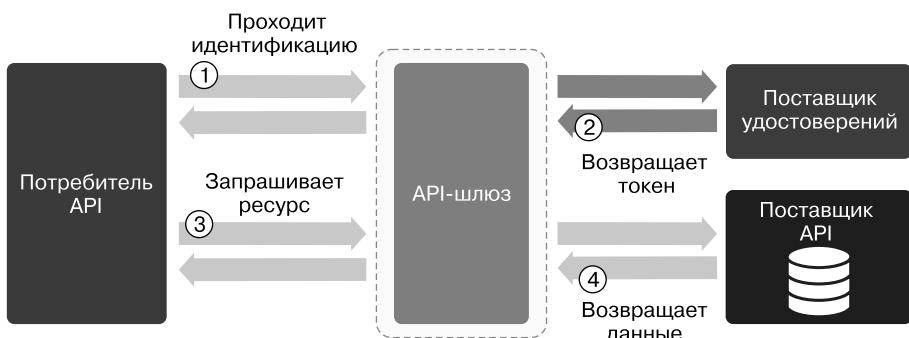


Рис. 7.9. Абстрактный поток безопасности API

Второй уровень безопасности находится в источнике, то есть на стороне, предлагающей данные. Поставщики API в этой модели пользуются контекстом, предоставляемым потребителем и шлюзом API, чтобы определить, возвращать ли данные и какие именно данные возвращать ④. Шлюз API в этой модели может добавлять и передавать поставщику API дополнительные метаданные: информацию об области, времени первой аутентификации и т. д. Для дополнительного контекста некоторые поставщики используют термин «шаблоны отображения» (<https://oreil.ly/kanWm>). Если шлюз API не поддерживает эту модель, вы можете рассмотреть вариант создания небольших микросервисов, чтобы

¹ Чтобы понять, как аутентификация работает с OAuth 2.0, см. это руководство (https://oreil.ly/Alm_M) на сайте Apigee.

обеспечить этот расширенный контекст. NGINX (<https://oreil.ly/-ndu>) рекомендует размещать прокси или микросервисы перед шлюзами API для выполнения аутентификации и проверки трафика, что приводит к оптимизации трафика, улучшению контроля и сокращению затрат.

Небольшие микросервисы в этой модели действуют как легкие прокси. Поставщики API здесь могут использовать приложение соглашения о совместном использовании данных в качестве точки принятия решения о политике. Они могут обратиться к нему, чтобы определить, есть ли у потребителя доступ. Кроме того, они могут использовать свой локальный контекст и бизнес-правила для принятия решений.



При использовании комбинации из шлюзов и поставщиков API описанная модель аутентификации может немного измениться. В некоторых случаях поставщики идентификации не отделяются шлюзами API, а вызываются напрямую. Для некоторых вызовов API дополнительная информация может быть получена из других конечных точек авторизации. Или может потребоваться комбинация токенов доступа.

Общей проблемой проверки и определения безопасности в источнике является то, что часто нужен дополнительный контекст. Пример — раздельное администрирование контрактов и клиентов. Механизм администрирования клиента хранит информацию о том, какие сущности могут просматривать учетные записи других клиентов. Чтобы запросить контекст из разных областей, вы можете использовать несколько разных подходов.

- Потребитель API «кэширует» дополнительный контекст и использует его при вызове следующих API. В случае с механизмом администрирования клиента и контракта потребитель API сначала вызовет механизм администрирования клиента, который вернет список сущностей. Следующий вызов извлечет данные с помощью API из механизма администрирования контракта. Список сущностей используется как дополнительный контекст для вызова этого API. Такая модель предпочтительна, но она работает только тогда, когда все области полностью доверены.
- Шлюз API знает, что некоторым API требуется дополнительный вызов для получения контекста. Apigee, например, называет это политикой вызова сервиса (<https://oreil.ly/Muyvm>). В этом случае шлюз API вызывает и собирает контекст из других областей и выполняет упрощенную оркестрацию. В целом этот метод более безопасен, чем предыдущий, потому что вы можете управлять вещами на стороне клиента. Таким образом, использование этого подхода тоже зависит от надежности областей-потребителей. Преимущество данного подхода в том, что шлюз API может кэшировать контекст, поэтому,

если будет сделан другой вызов, контекст не нужно будет снова получать. Недостаток — шлюз API может предоставить неправильный контекст при изменении атрибутов безопасности. Итак, в динамической среде лучше обновлять контекст при каждом вызове API. Обычно области знают, какой подход будет уместен в конкретных случаях использования.

- Предметные области используют идентификационный токен потребителя API для получения дополнительного контекста. Шлюз API в этой модели действует как посредник. При таком подходе все соответствующие области должны использовать одного и того же поставщика удостоверений. Эта модель работает только в том случае, если токен не обновляется регулярно, поэтому она не подойдет тогда, когда новые токены передаются обратно после каждого вызова API.
- Область напрямую вызывает другую область для получения контекста после возврата вызова API. Недостатком здесь может быть эффект пинг-понга, когда постоянно выполняются новые вызовы API. В этом подходе нет ничего плохого, но использование шлюза API будет лучшим вариантом.

Предпочтения вашей модели безопасности будут зависеть от используемых технологий и уровня доверия. Я видел, как некоторые крупные предприятия отделяют уровень безопасности от архитектуры API, развертывая специальный продукт или компонент безопасности в архитектуре как дополнительный уровень. IBM DataPower Gateway (<https://oreil.ly/R9WBz>), например, представляет собой платформу безопасности, которая более эффективно обрабатывает все дополнительные настройки безопасности, такие как проверка содержимого сообщений, IP-адресов (функциональность брандмауэра), ответов, токенов (JWT) и преобразование протоколов.

Задержка тоже может повлиять на выбор модели безопасности. Если запросы API должны быть обработаны в течение короткого периода времени, дополнительные уровни безопасности могут вызвать проблемы, потому что каждый переход в сети обычно увеличивает время ожидания. Вызов областей напрямую может быть единственным решением. Обычно я рассматриваю это скорее как исключение, чем как передовую практику.

Типичная проблема безопасности API — это интегрированные представления тогда, когда необходимо объединить много данных и логику интеграции. Например, для непрерывного предоставления данных определенной формы нужно несколько вызовов различных сервисов. В таких ситуациях *сервис агрегации* (рис. 7.10), объединяющий несколько сервисов и возвращающий их как новый сервис, может помочь избежать переговоров между потребителями и поставщиками. Такие переговоры могут повлиять на производительность. Для этих типов сервисов важно строгое соблюдение управления данными.

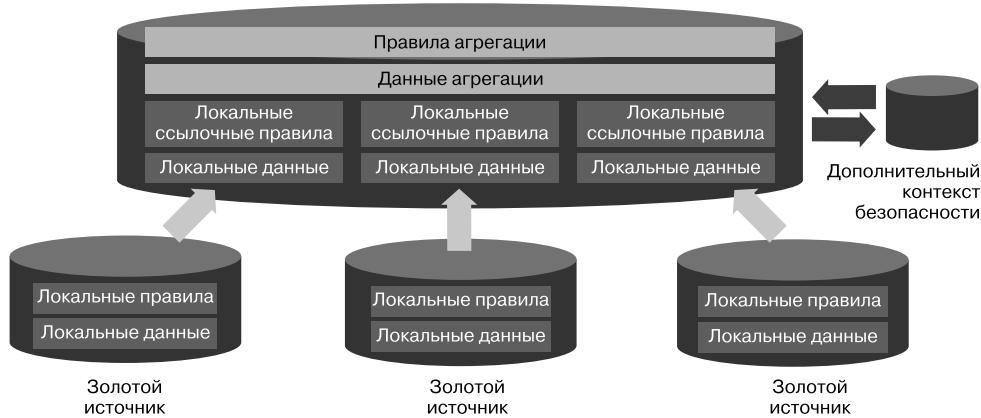


Рис. 7.10. Сервисы агрегации объединяют API от нескольких поставщиков для создания новых сервисов

Первое правило в таких ситуациях — все локальные данные и требования безопасности остаются собственностью исходных поставщиков. Агрегатор не может изменять данные и правила. Второе правило — сервисы агрегации берут на себя ответственность за все общие правила, включая вновь созданные данные¹. Есть одно общее правило: определенная комбинация данных не может быть использована. Эти всеобъемлющие правила могут создаваться и поддерживаться только самой службой агрегации, ведь это единственное место, где хранятся комбинации данных. Последнее правило состоит в том, что потребители данных ответственны за предоставление любого контекста, в котором (агрегированный) поставщик сервисов должен быть авторизован. Общие правила могут быть сложными, поэтому может потребоваться много контекста.

Потоковая архитектура

Контроль доступа на основе политик в потоковых и событийно-ориентированных архитектурах сложен. В настоящее время нет готовых структур безопасности. Kafka, например, поддерживает только RBAC (https://oreil.ly/v_iKy) и не обеспечивает фильтрацию сообщений на основе политик с использованием комбинаций атрибутов. Поэтому, если вам нужна детальная модель безопасности ABAC, придется спроектировать ее самостоятельно.

¹ Кроме того, вы можете хранить общие правила в центральном хранилище метаданных безопасности, где хранятся все политики и атрибуты. Службы агрегации при такой настройке больше не являются точкой администрирования политики.

Один из подходов – это разработка уровня применения политики, который использует микросервисы для чтения событий, берет инструкции из приложения соглашения о совместном использовании данных или механизма политики и применяет их. Возвращаемые события сохраняются в новых темах, ориентированных на потребителя (рис. 7.11).

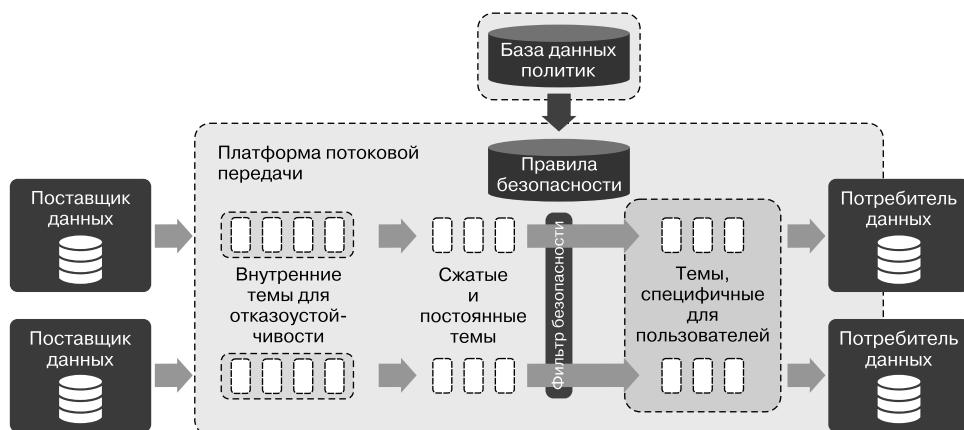


Рис. 7.11. Используя метаданные политик безопасности, заложенные в платформу, вы можете создавать детализированные фильтры и настраивать темы для конкретных потребителей

В Kafka события по мере поступления сначала сохраняются во внутренних темах. Если вы хотите дольше сохранять состояния приложения, то для этого могут использоваться сжатые темы. Следующий шаг – перенос всех политик безопасности и связанных метаданных на платформу потоковой передачи событий. Последний шаг – разработать платформу фильтрации сообщений на основе политик. Микросервисы, которые читают входящие данные, объединяют их с данными из политик, фильтруют данные и предоставляют темы, специфичные для потребителей, только с данными, доступными для потребителей. Эта структура фильтрации может объединять данные из других контекстов, если это требуется для принятия решений по политике: например, для получения стороны, обслуживающей клиентов, для конкретной финансовой транзакции из RDS.

Ключевые выводы из этих практических примеров в том, что существует множество вариантов. Проектирование и внедрение системы безопасности требует глубокого обдумывания того, какие компоненты использовать и как применить их к вашей общей архитектуре.

Интеллектуальный механизм обучения

В предыдущих разделах я упоминал *интеллектуальный механизм обучения* (*Intelligent Learning Engine, ILE*), который использует правила безопасности и принимает решения о том, какие данные разрешено видеть потребителям. Что я еще не обсуждал подробно, так это то, как правила безопасности создаются и поддерживаются в архитектуре.

Вначале количество правил будет ограничено. Сотрудники отдела безопасности знают, какие опасные комбинации привилегий потребители не могут использовать и какие данные безопасны. Но по мере роста количества атрибутов модель становится сложной и трудной в управлении из-за множества правил. Вот где в игру вступает ILE. Он управляет правилами безопасности в любом масштабе.

На начальном этапе механизм ILE использует технику обучения с учителем в фоновом режиме и проверяет решение сотрудников службы безопасности. Он изучает и дает рекомендации, но сотрудники службы безопасности все равно проверяют результат. И если они не установили политику безопасности, мы всегда можем прибегнуть к решениям, принятым владельцами данных, и начать учиться у них.



В машинном обучении используются термины «с учителем» и «без учителя». Модели обучения с учителем используют маркированные данные. Алгоритм учится предсказывать, так как он берет выходные данные из входных. При обучении без учителя модели используют немаркированные данные. Алгоритм может обучаться только на структуре входных данных.

Позже роли меняются местами: сотрудники службы безопасности подтверждают данные, предложенные интеллектуальным механизмом обучения. По мере того как количество правил будет расти, сотрудник службы безопасности постепенно будет играть роль аудитора. Аудиторы отбирают общее состояние системы безопасности для мониторинга и проверки. Со временем ILE может быть расширен до отчетов о тактике, методах и процедурах (*tactics, techniques and procedures, TTP*), определенных типах подозрительных действий, таких как вредоносное ПО, сбои и целевые атаки¹. На основе обнаруженных фактов несанкционированного доступа могут быть присвоены дополнительные оценки риска, чтобы сделать их еще более конфиденциальными. Эти сигналы и оценки должны контролироваться сотрудниками службы безопасности.

¹ Книга The Definitive Guide to Cyber Threat Intelligence (<https://oreil.ly/XzsF5>), написанная Джоном Фридманом (Jon Friedman) и Марком Бушаром (Mark Bouchard), содержит дополнительную справочную информацию о ДТС.

Итоги главы

Управление данными и их безопасность тесно взаимосвязаны. Управление данными — важнейший фактор успеха в создании современной практики обработки данных. Чтобы соответствовать строгим правилам, важно определить право собственности и установить стандарты для происхождения, классификации, цели и описания. Для этих задач также важно идентифицировать все приложения и уникальные (золотые) источники в организации. Чтобы сделать управление данными масштабируемым, нужно определить и верно назначить связанные роли и обязанности. Начните с малого и стремитесь к быстрым победам.

Для безопасности данных важно создать сквозную структуру, которая эффективно поддерживает процесс выполнения запроса данных. Эта структура — начало улучшенной модели безопасности на уровне данных. Она должна быть разумной частью вашей общей архитектуры. Продвигаясь вперед, постепенно расширяйте ее, добавляя больше метаданных, классификаций, меток, атрибутов и т. д. Чтобы еще больше масштабировать структуру, вы можете сделать ее интеллектуальной с помощью машинного обучения.

В следующей главе мы сосредоточимся на превращении данных в ценности. Вы узнаете больше о хранилищах данных области, конвейерах данных, моделях самообслуживания, бизнес-аналитике и продвинутой аналитике.

ГЛАВА 8

Превращение данных в ценность

Ранее вы узнали, что нужно для обеспечения защищенного и контролируемого доступа к данным. В этой главе рассказывается, как превратить данные в ценность. Это самая сложная часть.

Бизнес-требования всегда находятся на первом месте. Для того чтобы превратить данные в идеи или действия, нужно понимать, как движется информация, и использовать эти сведения для определения деловых возможностей. Сценарии использования могут начинаться как разовые проекты, но в идеале вы превратите свои ключевые данные в обслуживаемые решения, которые будут приносить постоянную пользу организации. В зависимости от требований бизнеса вы можете использовать разные методы: бизнес-аналитику, принятие решений в реальном времени или машинное обучение.

Есть еще нефункциональные требования. Чтобы удовлетворить большое количество сложных сценариев использования, нужно применить множество различных технологий БД. Вы можете выбрать один или несколько вариантов использования. Эти варианты могут отличаться типами преобразований, быстродействием, наличием возможностей параллельной обработки и моделями потребления, влияющими на конечный результат. Наконец, из-за ограничений производительности вам придется дублировать или реструктурировать данные для более быстрого чтения.

Предлагаемый подход для решения всех этих проблем — создание стандартизованных многоразовых шаблонов и архитектурных блоков. Они будут построены на основе распределения данных, обсуждаемого в главе 6, и будут работать с моделью управления, описанной в главе 7. К концу этой главы вы познакомитесь с различными моделями потребления, различиями между самообслуживающимися и управляемыми данными, нефункциональными аспектами выбора хранилищ данных, а также архитектурными блоками и принципами бизнес-аналитики и продвинутой аналитики.

Модели потребления

Перед тем как мы приступим к решению проблем с данными с помощью БД и инструментов, проанализируем две модели потребления, чтобы лучше понять проблему интеграции.

Прямое использование хранилищ данных только для чтения

Первая модель потребления напрямую использует хранилища данных только для чтения (read-only data store, RDS). Как описано в главе 3, RDS нужны для многократного обслуживания потребителей больших объемов неизменяемых данных. Благодаря производительности и шаблонам доступа они могут идеально подходить для прямой отчетности, специального анализа, аналитики и самообслуживания. В этом шаблоне данные только читаются — новые не создаются. Приложения-потребители используют RDS напрямую как источники данных и могут выполнять упрощенную интеграцию, основываясь на сопоставлениях между схожими элементами данных. Они могут изменять контекст и временно создавать новые данные, но не требуют, чтобы результаты сохранялись в другом месте. Большое преимущество этой модели в том, что она не требует создания новых моделей данных. Вы не извлекаете, не преобразуете и не загружаете данные в новую базу. Преобразования происходят быстро, а результаты не требуется сохранять в новом месте.

Эта модель прямого потребления требует, чтобы RDS обеспечивали адекватную производительность, поэтому интеллектуальные и потребляющие приложения должны иметь возможность обрабатывать данные напрямую. Модель потребления на высоком уровне проиллюстрирована на рис. 8.1.

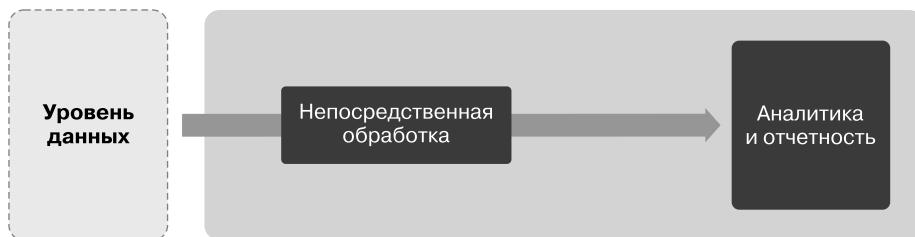


Рис. 8.1. RDS используются непосредственно в качестве источника данных для приложений-потребителей. Потребители выигрывают от такой схемы потребления, так как им не приходится заботиться о сложностях извлечения, преобразования и загрузки

Проблема здесь в том, что нужды потребителей могут превосходить то, что предлагают RDS. В некоторых случаях действительно нужно создавать новые данные: например, когда результаты сложных преобразований обрабатываются аналитическими моделями, генерирующими новые бизнес-идеи. Чтобы сохранить эту бизнес-информацию для последующего анализа, необходимо поместить ее куда-нибудь, например в базу данных. Другая ситуация может заключаться в том, что объем данных, которые необходимо обработать, превышает то, что может обработать платформа RDS. В таком случае объем обрабатываемых данных, например исторических данных, настолько велик, что может понадобиться поэтапный перенос данных в новое место, их обработка и предварительная оптимизация для последующего использования. Еще один вариант — необходимость объединить и согласовать несколько RDS. Обычно это требует согласования множества задач. Долгое ожидание, пока все эти задачи выполняются, может отрицательно сказаться на пользовательском опыте. Все эти последствия подводят нас ко второй модели потребления данных: созданию хранилищ данных домена.

Хранилища данных предметной области

Нам нужно более тщательно управлять вновь созданными данными. Для этого и предназначены хранилища данных предметной области (domain data store, DDS). Их роль заключается в хранении вновь созданных данных и поддержке вариантов их использования потребителями (рис. 8.3). Этот строительный блок обычно разрабатывается как автономная база данных, оптимизированная для пользователей и приложений, но также может быть многовариантной моделью хранения (<https://oreil.ly/ji3D2>) или конгломератом данных RDS и интегрированных данных.

DDS как строительный блок, показанный на рис. 8.2, остается технологически независимым, чтобы упростить поддержку как можно более широкого спектра вариантов использования. Для создания DDS потребителям необходимо представить ряд возможностей, таких как инструменты ETL, различные технологии баз данных, инструменты планирования, CI/CD и т. д. Позже мы еще вернемся ко многим из этих аспектов.

В архитектуре DDS находятся между уровнем данных и приложениями-потребителями, такими как бизнес-аналитика и аналитические инструменты. Чтобы предотвратить дублирование бизнес-логики (интеграции) или создания слишком большого количества зависимостей, важно установить для DDS четкие границы — ограниченные контексты, в которых данные интегрируются для конкретной бизнес-возможности.

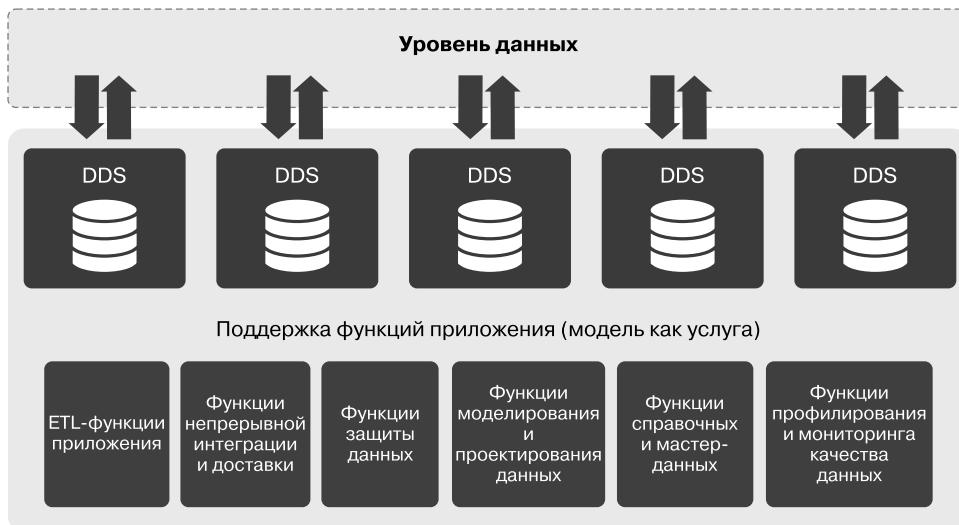


Рис. 8.2. Сложность базовой инфраструктуры скрыта для команд предметных областей.
Интеграцию данных из различных областей предпочтительнее выполнять с помощью
управляемых и централизованных архитектурных блоков платформы

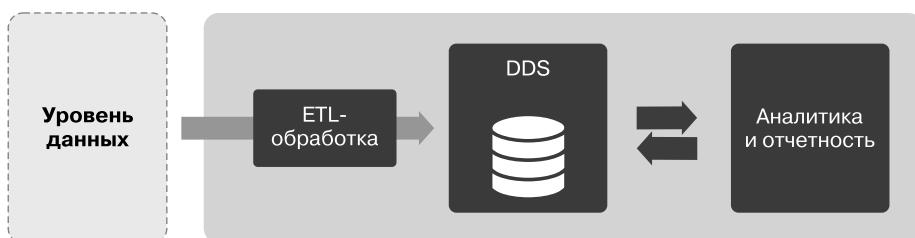


Рис. 8.3. Основная цель DDS — поддержка интеграции и создания новых данных. DDS необходимы только тогда, когда потребители не могут использовать RDS напрямую. Базовая технология выбирается в зависимости от варианта использования исходя из затрат, гибкости, знаний и навыков, а также нетехнических требований, таких как производительность, надежность, структура данных, доступ к данным и шаблоны интеграции

Определение объема, размера и размещения логических границ DDS затруднительно и вызывает проблемы при распределении данных между областями. Обычно ограниченные контексты предметно ориентированы и соответствуют как бизнес-возможностям, так и потокам ценностей (см. подраздел «Бизнес-архитектура» на с. 49). При определении логических границ области имеет смысл разложить ее на подобласти для упрощения операций моделирования и внутреннего распределения данных внутри самого домена.

Разбиение области особенно важно, когда она большая или когда для подобластей требуется общая — повторяемая — логика интеграции. В таких ситуациях может помочь выделение общей подобласти, обеспечивающей логику интеграции таким образом, чтобы позволить другим подобластям стандартизировать и извлекать из нее выгоду¹. Основное правило — сохранять модель, общую для подобластей, небольшой и всегда согласованной с единым языком.



Роль DDS могут играть транзакционные или операционные приложения, потому что они часто используют и интегрируют небольшие части данных. Для распространения этих данных всегда должен использоваться уровень данных.

Границы DDS определяют разделы ответственности за данные. К ним относятся качество данных, владение, интеграция и распространение, моделирование и безопасность. Одна из этих ответственостей включает обмен интегрированными данными с другими потребителями. В этом случае для распространения данных можно будет использовать тот же уровень данных (мы вернемся к этому позже, в разделе «Распространение интегрированных данных» на с. 277, а также в главе 9).

Способы проектирования и управления DDS различаются для разных вариантов использования из-за вовлечения разных целевых групп пользователей, бизнес- и нефункциональных требований. Таким образом, каждое хранилище DDS будет иметь дополнительные инструменты и целевую операционную модель.

Целевая операционная модель

У многих организаций есть свои бизнес-аналитики (business intelligence, BI) и аналитические инструменты. Эти инструменты начали появляться, когда стали популярными корпоративные хранилища данных. Большинство из них плотно работают с хранилищами данных и обслуживаются одними и теми же командами.

Из-за отсутствия конкретных аналитических возможностей и необходимости повышения скорости бизнес-пользователи сами начали покупать и развертывать собственные инструменты. Они создали двухточечные интерфейсы и подклю-

¹ В предметно-ориентированном проектировании используется термин «общее ядро», чтобы указать, что часть модели области совместно используется разными командами или подобластями. Стратегия интеграции с общим ядром снижает дублирование и на-кладные расходы.

чили аналитические инструменты напрямую к хранилищам и операционным системам. Они начали извлекать, преобразовывать и загружать данные в свои самоуправляемые бизнес-окружения. По мере появления новых вариантов использования и возможностей все больше этих инструментов применялось для решения только отдельных проблем, без учета их связанности. Вследствие применения такого разнообразия инструментов архитектура стала громоздкой, сложной в управлении и более дорогой.

Чтобы бизнес-пользователи не привели архитектуру в хаос, требуется изменить модель. Нужно создать контролируемую среду, которая глубоко интегрируется в базовую архитектуру управления данными и поддерживает децентрализованную модель множества бизнес-областей, а также удовлетворяет потребности разных аудиторий и групп пользователей на основе *самоподдержки*. Бизнес-пользователям нужна скорость. Если, например, интеграция, распространение и потребление пользовательских данных осуществляются вручную, то архитектура получится немасштабируемой. Бизнес-пользователи попытаются решить свои проблемы иначе. Вместо того чтобы заставлять одних пользователей ждать других, новая архитектура должна «дать возможность» областям выполнять работу самостоятельно.

Самообслуживание, которое позволяет пользователям легко запрашивать доступ к данным и инструментам, влияет на возможности и решения интеграции. Самообслуживание — это автоматизация, метаданные и интеграция. Прежде чем предлагать какие-либо решения, давайте разберемся с целевой аудиторией.

Специалисты по данным как целевая группа пользователей

Давайте рассмотрим четыре типа специалистов в области данных: аналитиков данных, специалистов по обработке данных, инженеров по обработке данных и конечных пользователей. Все они обычно работают в тесном сотрудничестве для достижения целей бизнеса. Они могут быть частью одной бизнес-команды или входить в состав центральной экспертной группы. Преимущество централизованной группы обработки данных в том, что специалисты могут обмениваться техническими навыками и применять передовой опыт. С другой стороны, понимание рыночных тенденций, целей и стратегий бизнеса важно для превращения данных в ценность. Многие крупные организации приняли гибридные модели, сочетающие в себе преимущества централизованных групп со знанием специфики бизнеса.

Эти профессиональные роли в области данных требуют разного опыта, знаний и навыков. Их понимание поможет вам лучше понять, что нужно этим ролям.

- *Аналитики данных* обычно знакомы с бизнес-процессами и операциями компании. Они знают, какие вопросы есть у бизнес-пользователей. Обычно они имеют общее представление о SQL и более знакомы с отчетами и инструментами бизнес-аналитики. Они не сильно разбираются в аналитических моделях и алгоритмах, но знают, какие переменные и функции применяются в этих моделях. Они пользуются доверием руководства и конечных пользователей бизнеса. Они умеют хорошо общаться. Аналитиков данных обычно называют опытными пользователями, потому что они знают, как анализировать данные с помощью инструментов, но не обязательно являются специалистами по данным или экспертами в области бизнес-аналитики.
- *Специалисты по обработке данных* — это статистики, имеющие опыт работы в области компьютерных наук или математики. Они знакомы с алгоритмами, машинным и глубоким обучением. Они умеют программировать и использовать такие языки, как Python, Scala, R и Java. Обычно они знакомы с бизнесом и следят за тенденциями рынка. Их конечная цель — найти закономерности и корреляции и сделать прогнозы на основе данных.
- *Инженеры по обработке данных* — это пользователи, отвечающие за сбор, интеграцию и обогащение данных. Они важны, так как позволяют аналитикам и специалистам по обработке данных правильно выполнять свою работу. Ожидается, что они будут знать необработанные данные, SQL и иметь возможность экспортить данные и преобразовывать их в нужный формат. Инженеры по обработке данных также добиваются успеха в средах, управляемых ИТ. После того как эксперименты подтвердят ценность процесса, инженеры могут начать автоматизировать его, чтобы данные собирались, очищались, интегрировались и становились доступными. В зависимости от целевой операционной модели инженеры по обработке данных могут выполнять операции самостоятельно (DevOps) или тесно сотрудничать с разработчиками и администраторами платформы.
- *Конечные пользователи* — это аналитики и менеджеры, которые в конечном итоге будут принимать решения. Они потребляют продукцию, предоставляемую аналитиками и специалистами данных, а также инженерами по обработке данных. Эти результаты могут быть представлены в виде отчетов, обзоров или сложных визуальных панелей мониторинга с диаграммами, основанными на ETL и моделях.

Чтобы позволить специалистам по обработке данных самим продвигать инициативы, нужны инструменты. В следующих разделах я опишу самые важные инструменты и конкретизирую требования.

Самая большая проблема предоставления ценности в масштабе предприятия в том, что многие инструменты бизнес-анализа и аналитики разбросаны по всей

организации. Все они пытаются обратиться к различным аспектам решения проблемы науки о данных. Кроме того, отсутствует полностью интегрированный опыт, ведь инженерия данных чаще всего отделена от разработки аналитических моделей. Наконец, сценарии вариантов использования разнообразны и для их реализации необходимо большое количество разнообразных хранилищ данных. Некоторые данные поступают быстро, в то время как другие — медленно, большими порциями, в том числе исторические данные. Невозможно обеспечить общий способ интеграции и проектирования новых хранилищ данных для будущего, но есть несколько общих шагов, которые нужно выполнять всегда.

Бизнес-требования

Убедитесь, что бизнес-цели и задачи четко определены, детализированы и полны. Их понимание — основа вашего решения. Для этого вы должны уточнить, какие бизнес-задачи необходимо решать, какие источники данных потребуются, какие решения должны работать, какая обработка данных должна выполняться в реальном времени или в автономном режиме, какие требования к целостности имеются и какие результаты будут повторно использоваться другими областями.

Бизнес-требования будут определять объем данных, которые необходимо получить. Например, чтобы разработать алгоритм машинного обучения на данных с высокой изменчивостью, вам, вероятно, понадобятся гораздо более подробные данные, чем для простой модели с низкой изменчивостью¹. Всегда полезно начинать сверху вниз, исходя из бизнес-требований.

Нефункциональные требования

Следующий шаг построения решения — определение нефункциональных требований. К ним относятся требований к затратам, масштабируемости и производительности, задержке, простоте обслуживания, объему, разнообразию, скорости, согласованности, безопасности и управлению, характеристикам записи и чтения и т. д. Каждое из этих требований укажет новое направление для анализа. Для всех этих решений возникает одна дилемма: какую технологию хранения данных следует использовать? Выбор оптимального хранилища зависит от многих критериев. Палочки-выручалочки не существует. Например, простое онлайн-приложение может прекрасно работать либо с хранилищем типа «ключ — значение», либо с RDBM. Иногда это дело вкуса и опыта.

¹ В машинном обучении существует общий принцип: большой объем данных лучше самых умных алгоритмов (<https://oreil.ly/ntW42>).

ЧТО СЛЕДУЕТ УЧЕСТЬ

- Как структурированы данные? Структура влияет на этапы предварительной обработки, тип очистки, этапы моделирования, тип хранилища и многое другое. Непонимание этого с самого начала приведет к потере времени и неверному результату.
- Предсказуемы ли запросы? Это может иметь большое значение. Предсказуемые запросы можно оптимизировать, например, с помощью кэшей, индексов или предварительно оптимизированных данных.
- Какие типы запросов используются? Простой поиск, объединение, соединение, математический, текстовый или географический поиск, другие сложные операции? Например, реляционные базы данных обычно лучшеправляются с соединениями.
- Каковы требования к текущим, историческим и архивным данным? Все ли их нужно хранить?
- Как сохранить баланс между оптимизацией целостности и производительностью? Например, должно ли приложение обеспечивать строгую согласованность или достаточно согласованности со временем? Если целостность важна, RDBM обычно лучше обеспечивают согласованность.
- Какие компромиссы можно найти в отношении характеристик чтения, записи и целостности данных? Разные модели данных имеют разные характеристики. Хранилище данных, например, имеет гибкую конструкцию и может обрабатывать параллельные нагрузки, но чтение данных требует больших затрат производительности.
- Какие операции будет выполнять база данных? К некоторым видам операций нужен особый подход. Распределенные файловые системы — хороший выбор для добавления, но не для вставки (произвольного доступа). В противном случае лучше использовать реляционные БД.
- Какого размера и с какой скоростью данные поступают и уходят? Некоторые базы данных NoSQL плохо справляются с пакетной загрузкой. Для высокоскоростного приема данных (обмен сообщениями и потоковая передача) нормализованные модели данных и модели согласованности ACID могут снизить производительность из-за контроля целостности и блокировки.
- Какие протоколы доступа к данным нужны? Некоторые базы данных доступны только через SQL, ODBC/JDBC или собственные движки, в то время как другие разрешают доступ только через REST API.
- Насколько гибко нужно адаптировать или изменить структуру данных? Базы данных NoSQL могут хорошо справляться с изменением структур данных, поскольку они могут быть «бессхемными».
- Какая масштабируемость и эластичность потребуется? Некоторые системы поддерживают динамическое горизонтальное масштабирование.
- Другие соображения включают стоимость, стандарты с открытым исходным кодом, расширенный функционал, интеграцию с другими компонентами, безопасность (например, контроль доступа), конфиденциальность, управление данными и простоту разработки и обслуживания.

Я рекомендую выбирать, сколько и какие хранилища данных вы хотите предложить организации. Возможно, вы захотите определить общий набор технологий БД многократного использования или хранилищ данных и шаблонов, чтобы каждое хранилище данных было максимально эффективным. Критически важным и промежуточным приложениям можно разрешить работать только со строгими моделями согласованности, а бизнес-аналитика и отчетность могут быть разрешены только в хранилищах, которые обеспечивают быстрый доступ SQL. Не будет лишним и выбрать локальную и облачную среды. В конечном итоге вы получите список из нескольких технологий баз данных и хранилищ, способных поддержать большинство вариантов использования.

Построение конвейера данных и модели данных

Третий шаг — проектирование конвейера данных. Он будет отбирать данные и объединять их в целевую модель. Важно здесь то, что все данные уже сохраняются в хранилищах данных только для чтения, поэтому нет необходимости делать дополнительную копию. Это означает, что вы либо получаете данные по обычному графику, либо ждете, пока поставщик данных сообщит вам, когда можно начинать обработку. При переносе данных из одного контекста в другой обычно требуется преобразование. Для преобразования контекста вам понадобится инструментарий ETL (который мы обсудим позже).

Конвейер, принимающий данные в режиме реального времени, работает иначе из-за необходимости захвата и сохранения сообщений также в реальном времени. Вы можете сохранить входящие сообщения в папке или базе данных для дальнейшей обработки. Другой вариант — задействовать буфер или дополнительный компонент приложения либо воспользоваться возможностями интеграции потоковой платформы, которые позволяют анализировать, обрабатывать и распространять сообщения дальше.

Я проиллюстрировал ETL и варианты потоковой передачи в архитектуре высокого уровня на рис. 8.4. Обратите внимание, что в зависимости от ваших требований возможно множество вариантов. Например, вы можете перемещать данные как есть перед преобразованием или применением целевой семантики. Вы также можете комбинировать пакетную обработку ETL и потоковую обработку. Вы даже можете реализовать многовариантную конструкцию, используя разные технологии хранения данных для удовлетворения различных потребностей.



Рис. 8.4. Для обработки данных из хранилища данных, доступного только для чтения, используется пакетная обработка; для реального времени — потоковая

Создание конвейера данных — самая сложная часть проекта. Начните с малого и сохраняйте гибкость. Большинство неудач при проектировании случается, когда инженеры пытаются объять необъятное и смоделировать все данные сразу. Еще одна распространенная ошибка — это сразу броситься в технологию и начать писать код, не продумав требования. Прежде чем приступить к сборке, тщательно продумайте все функциональные и нефункциональные требования, а также рассмотрите фундаментальные вопросы, обсуждаемые во врезке «Что следует учесть» выше. Нужно ли многократное использование? Как лучше всего обрабатывать все данные? Есть ли зависимости и можно ли выполнять несколько шагов одновременно?



Специалисты по анализу данных любят ссылаться на то, что они называют правилом 80/20: 80 % ценного времени специалиста по данным тратится на поиск, очистку и систематизацию данных, а на фактическую разработку уходит только 20 %. Не существует единого алгоритма, который использует необработанные данные и дает лучшую модель.

Я советую собрать конвейеры данных в виде изолированной серии неизменяемых преобразований. Так их можно будет повторно использовать и легко комбинировать. В этом случае ввод, логика преобразования и выход должны быть четко отделены друг от друга. Для получения воспроизводимых выходных данных я рекомендую организовать управление версиями во всех конвейерах. Чтобы повысить производительность, вы можете спроектировать конвейеры для параллельной работы, пользуясь гибкостью современной инфраструктуры. Когда дело доходит до сложного преобразования и обработки данных, иногда лучше использовать механизмы обработки в памяти и распределенные механизмы обработки, такие как Apache Spark. Еще одно соображение — написать логику преобразования на языке программирования высокого уровня в экосистеме, удобной для разработки.

СРАВНЕНИЕ КОНВЕЙЕРОВ SQL И NOSQL

Как сделать правильный выбор между конвейерами SQL и NoSQL?

Конвейеры SQL требуют лучшего понимания отношений между таблицами, так как они эффективнее обрабатывают и выполняют сложные запросы. Поэтому конвейер SQL может потребовать больше шагов проверки и логики преобразования и обновления, поскольку данные могут быть проанализированы и собраны в правильную (ограничивающую) структуру.

С другой стороны, конвейеры NoSQL часто получаются более простыми и гибкими. Они могут создавать данные немедленно и динамически, без определения структур. Кроме того, их легче масштабировать: запросы могут выполняться быстрее, поскольку они не связаны с соединением множества таблиц. Данные легче копировать по горизонтали; но реализация обработки запросов и интеграция могут оказаться намного сложнее.

В отношении качества данных можно положиться на проверки в RDS, но это не означает, что в конвейерах нет дополнительных проверок. Мнения потребителей о качестве данных различаются, поэтому конвейеры следует расширять для решения общих проблем ETL, которые обычно связаны с ограничениями, полнотой, правильностью и чистотой данных. Кроме того, нужно осторожно относиться к вопросам безопасности и конфиденциальности. Поэтому перед импортом и обработкой данных ожидаются дополнительные действия.

Наконец, весь конвейер должен быть основан на метаданных с информацией о происхождении. Иначе говоря, на способности отслеживать и понимать, какие данные и как изменяются на каждом этапе конвейера. Это включает прозрачные обратные указатели на RDS и потоковые сообщения. Имена файлов и событий, бизнес-ключи, идентификаторы исходной системы и т. п. должны быть доступны в центральных инструментах, чтобы гарантировать правильность и полноту всех этапов преобразования. Возможно, вы захотите использовать каталог, чтобы объединить все архитектуры интеграции. Я рекомендую сформулировать строгие принципы создания конвейеров данных в командах, например определяя, при каких условиях информация о происхождении должна доставляться централизованно.

Не будет лишним объединить все продукты и фреймворки ETL в единое портфолио. Традиционные продукты предварительно настроены и могут автоматически удалять свои метаданные централизованно, в то время как для облегченных фреймворков может потребоваться использовать сценарии или дополнительные компоненты для передачи метаданных. Таблица 8.1 – хороший пример одной из таких категорий.

Таблица 8.1. Возможности интеграции данных можно условно разделить на традиционные, легкие и эластичные инструменты и фреймворки, ориентированные на анализ данных

Категории	Возможности	Типичные варианты использования и примеры
Традиционные инструменты ETL	<ul style="list-style-type: none"> • Возможность подключения к данным, слияние, объединение, поиск ссылок. • Использование множества для многих типов источников: СУБД, NoSQL, API, полуструктурированные (XML, JSON и т. д.) и неструктурированные (социальные сети, журналы). • Расширенные возможности определения происхождения, визуализации и отладки. • Встроенная документация по процедурам интеграции. • Среда проектирования на основе графического интерфейса, управляемого мышью. • Расширенный планировщик заданий, мониторинг и предупреждение об ошибках, обработка, журнализование, контроль версий, управление выпусками 	<ul style="list-style-type: none"> • Добавление сложных инструментов интеграции, таких как Informatica (https://www.informatica.com/) и Talend (https://www.talend.com/), для удовлетворения сложных требований. • Автоматическая доставка информации о происхождении в центральное хранилище. • Широкая поддержка различных источников. • Строгие требования к метаданным, отладке и аудиту. • Использование в больших командах и долговременных проектах. • Длинная кривая обучения и сложная адаптация
Легкие инструменты ETL	<ul style="list-style-type: none"> • Среда проектирования на основе графического интерфейса для перетаскивания, ориентированная на перемещение данных и (отчасти) простую интеграцию. • Сосредоточенность на конкретных случаях использования интеграции данных без охвата всего спектра источников и вариантов обработки. • Базовое планирование заданий, мониторинг, предупреждение об ошибках и ведение журнала. • Ограниченные возможности отладки или их отсутствие. • Ограниченные или отсутствующие возможности командной разработки, такие как контроль версий или управление выпусками 	<ul style="list-style-type: none"> • Более простой, современный и гибкий вариант с такими инструментами, как StitchData (https://www.stitchdata.com/) и Singer (https://www.singer.io/). • Более простые требования к интеграции. • Нацеленность на автоматизацию, а не на сложную интеграцию. • Более дешевая модель с оплатой по факту использования. • Эластичность; можно выполнять выгрузку из облака или распределенной системы. • Механизм обработки данных (платформы Big Data). • Более быстрое обучение

Категории	Возможности	Типичные варианты использования и примеры
Инструменты и фреймворки, ориентированные на программирование	<ul style="list-style-type: none"> Фреймворк (бессерверный) для программирования для интеграции данных на облачных платформах/платформах больших данных. Поддержка создания конвейеров данных с сочетанием простых функций визуальной компоновки и программирования. Поддержка проверки конвейеров данных на облачных платформах, исключая ручное программирование. Возможности отладки с помощью контрольных точек. Поддержка оркестрации конвейеров данных 	<ul style="list-style-type: none"> Использование в (сложной) инженерии данных и науке о данных. Использование бессерверных фреймворков или распределенных языков обработки данных, таких как MapReduce, Spark, Scala, Python и Java. Использование дополнительных компонентов или сценариев для передачи информации о происхождении¹. Управление такими функциями, как функции Azure, AWS Lambda и т. д. Добавление науки о данных, например машинного обучения. Интеграция данных со сложными преобразованиями. Долгая кривая обучения. Экономия на инструментах, но большие расходы на консультанта

Самая важная часть проектирования конвейера — моделирование данных в целевой схеме: моделирование и проектирование данных. Из-за того что мы отказываемся от корпоративных моделей данных и используем DataOps, как автоматизированную, ориентированную на процессы методологию, которая позволяет командам управлять собственными конвейерами и БД, важно дать правильные инструкции. Есть много возможных вариантов, но я приведу четыре наиболее распространенных.

- *Транзакционные хранилища данных.* Они очень похожи на системы OLTP, с той небольшой разницей, что они потребляют и интегрируют данные из других областей. Хотя транзакционные хранилища могут быть современными и работать с неструктурированными данными, ожидается, что они будут использовать строгую модель согласованности для удовлетворения требований к целостности. Для этого требуется строгое соблюдение схемы, а иногда

¹ Spline (<https://oreil.ly/VCXa1>), например, — это проект, который помогает инженерам получить представление об обработке данных и их происхождении в Apache Spark.

и более строгая нормализация⁴. Они могут предусматривать совместную работу с другими приложениями, поэтому обычно имеют обширный REST API.

- *Хранилища бизнес-аналитики.* Модели данных для бизнес-аналитики и отчетности часто бывают реляционными и многомерными. Данные обычно моделируются в виде фактов и измерений для дополнительного контекста. Самый популярный и простой выбор — это звездообразная схема или схема 3NF.
- *Хранилища аналитических данных.* Хранилища аналитических данных или файловые хранилища используются для аналитических моделей, таких как машинное обучение. Их цель — предоставить высококачественные данные для разработки точных моделей. Данные обычно слажены и денормализованы. Вы можете рассмотреть возможность разделения хранилища данных на подмножества: одно хранилище для данных обучающих моделей и одно для проверки и оценки данных по отношению к новым данным. Проблема разработки и развертывания в том, что для реализации моделей часто требуются дополнительные языки программирования, а для хранилища аналитических данных — дополнительные микросервисы и контейнерная инфраструктура. Этот момент будет рассмотрен в разделе «Возможности аналитики» на с. 283.
- *Гармонизированные хранилища данных.* Это вариант применим, когда область можно разделить на несколько подобластей, а требования к данным во многом пересекаются. Интеграция данных в согласованный уровень данных заранее может помочь избежать повторной работы и облегчить жизнь других команд и подкоманд. Эти хранилища используются для заполнения хранилищ бизнес-анализа и аналитики. Сложность интеграции растет экспоненциально с увеличением количества источников и подобластей. Поэтому я рекомендую интегрировать заранее не все данные, а только те, которые будут использоваться повторно. Кроме того, интегрированные данные можно проверить на наличие дубликатов для поддержания уникального набора основных удостоверений. В зависимости от того, нужно ли проверять данные, могут быть разработаны дополнительные уровни для сохранения любых данных. Обычные структуры — 3NF и хранилище данных. Наконец, данные могут загружаться постепенно, поэтому загружаются только новые данные или изменения для ранее загруженных.

Для всех вариантов хранилищ и проектов вы должны стандартизировать инструменты обработки метаданных, включая сбор метаданных о происхождении и метаданных, используемых при разработке концептуальных, логических и физических моделей данных. Вам также понадобится стандартизировать раз-

⁴ Схему можно применить и на уровне приложения. Это может быть нужно, когда базы данных без схемы NoSQL отказываются от контроля целостности.

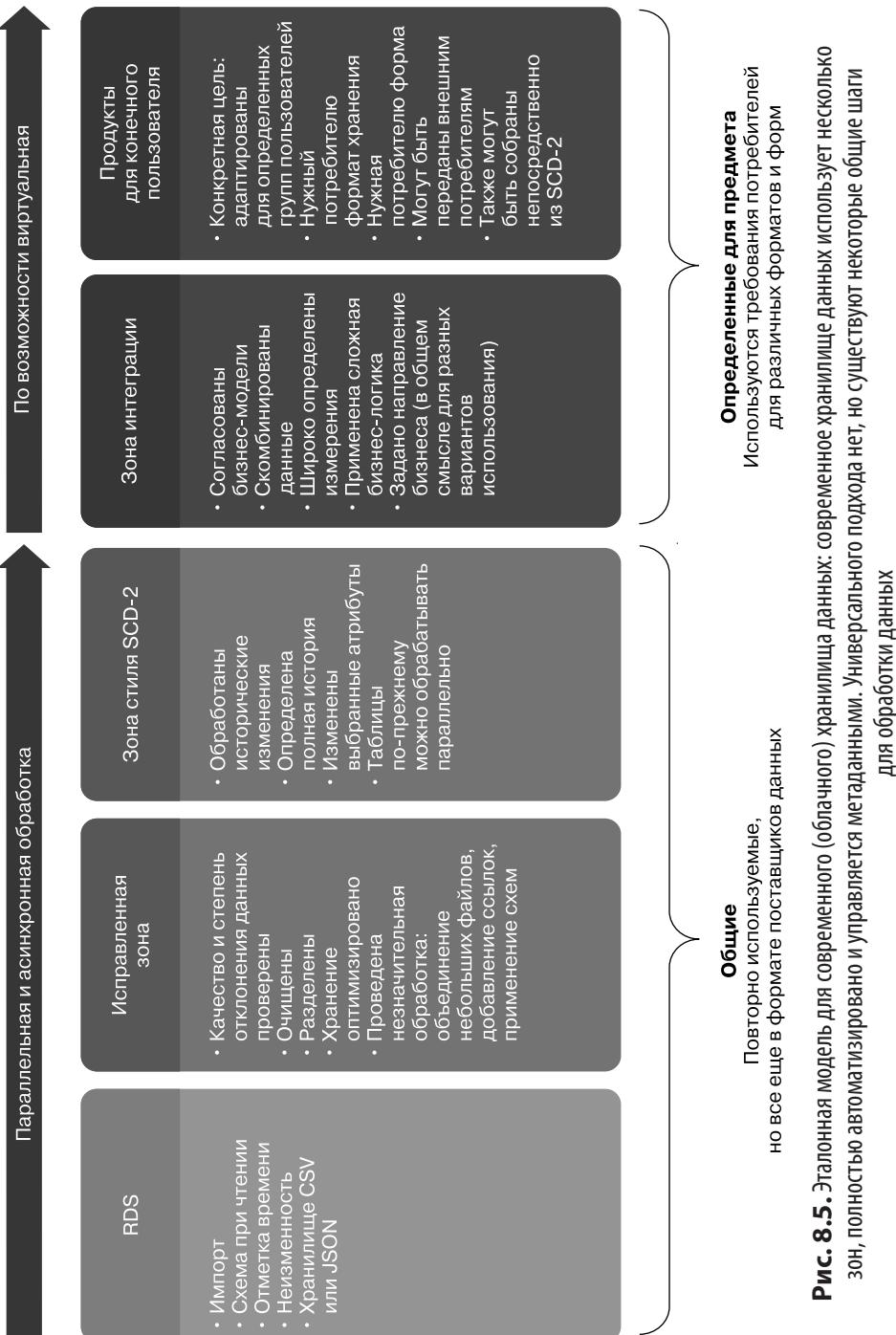
работку программного обеспечения, включая автоматизацию развертывания кода, оркестрацию и то, какие платформы, библиотеки и инструменты могут развертывать пользователи. Наконец, вам нужно будет стандартизировать моделирование и проектирование данных — заполнение, стандарты именования, материализацию, разрешения, методы нормализации данных и т. д.¹ Типичный подход к организации всего этого — использование внутреннего источника или создание главного центра экспертизы².

Разные хранилища данных организуют свои данные и управляют ими изнутри. Один из распространенных способов организации — разделить (логически или физически) задачи заполнения, очищения, обработки, гармонизации, обслуживания и т. д. В современных хранилищах данных такая организация может включать разные зоны (рис. 8.5) с использованием различных методов хранения — папок, корзин, баз данных и т. д. Зоны позволяют комбинировать цели, поэтому хранилище можно использовать, чтобы одновременно облегчить аналитики и операции. Границы всех хранилищ и зон должны быть четко определены. Они не должны охватывать несколько ограниченных контекстов. Каждая область, которая потребляет, интегрирует и создает новые данные, должна иметь собственное выделенное хранилище DDS с несколькими зонами.

История внутренней организации данных может усложняться, если область большая и состоит из нескольких подобластей. Хранилище DDS в этом представлении является более абстрактным: зоны могут совместно использоваться несколькими подобластями и могут быть исключительными. Попробую конкретизировать это на примере. Для большой области вы можете построить границу вокруг всех зон одного DDS. В этом DDS, например, первые две зоны могут совместно использоваться несколькими подобластями. Таким образом, очистка, исправление и накопление исторических данных обычно выполняются для всех подобластей. В случае преобразования ситуация становится сложнее, потому что данные должны быть специфичными для подобласти или варианта использования. Поэтому могут появиться общие конвейеры и конвейеры, определенные исключительно для одного варианта использования. Вся эта цепочка данных, включая конвейеры, принадлежит друг другу и может рассматриваться как одна гигантская реализация DDS. Внутри нее, как вы только что узнали, проводятся разные границы: общие для всех подобластей и конкретные для каждой подобласти.

¹ Степень отношения определяет, что именно представляет каждая строка. В таблице products («товары»), например, одна запись может соответствовать одному товару, поэтому каждый товар находится в отдельной строке — ровно одна строка для каждого товара. Убедившись, что все записи очищены, вы точно указываете, что содержится в строке таблицы. Это дает пользователям представление о том, какие данные можно объединить.

² Ник Тьюн (Nick Tune) в своем блоге (<https://oreilly.com/TY1xA>) объясняет, как лучше управлять межгрупповыми зависимостями и устранять их.



Распространение интегрированных данных

Как мы обсуждали в главе 2, из систем золотых источников разрешается доставлять только первоначально сгенерированные золотые данные. Но что произойдет, если интегрированные данные должны быть распределены между DDS в разных ограниченных контекстах? В такой ситуации потребитель данных становится поставщиком и должен следовать тем же принципам. Данные должны быть доставлены обратно на уровень данных, и потребитель должен соответствовать всем критериям поставщика. Эта модель изображена на рис. 8.6.

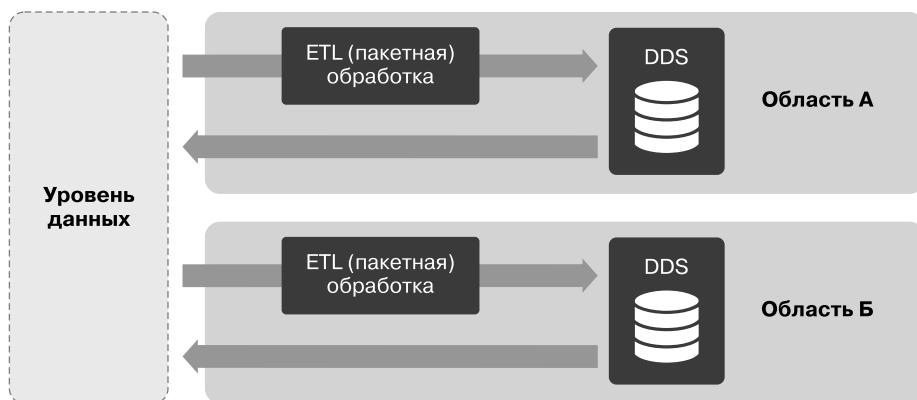


Рис. 8.6. DDS четко ограничены и разделены. При распределении данные должны быть доставлены обратно на уровень данных и должны соответствовать всем критериям поставщика данных. Следовательно, потребитель данных становится поставщиком

Я рекомендую различать золотые данные (см. подраздел «Золотой источник» на с. 42) и данные, интегрированные с помощью нескольких основных принципов. *Интегрированные данные* – это данные, которые были потреблены, объединены и преобразованы в новый контекст¹. Входные и оригинальные данные возникли где-то еще, поэтому происхождение очень важно – оно показывает все зависимости от других областей. Право собственности на данные работает иначе, поскольку ответственность за создание и качество данных выходит за рамки интеграции. Таким образом, для распространения интегрированных данных необходимо одобрение владельца (-ев) оригинальных данных.

Наконец, распространение интегрированных данных создает новые зависимости, поэтому подумайте о том, чтобы разбить данные на небольшие узкоспециализи-

¹ Некоторые инженеры отрасли используют термин «подготовленные данные», чтобы дать понять, что необработанные данные были обработаны.

рованные части, которые решают только одну конкретную задачу. Ограничте общую предметную модель до минимума. Не создавайте распределенный монолит, на который полагаются все команды, но требуйте очень четкого определения масштабов. Если данные подлежат интенсивному повторному использованию, они должны стать частью дисциплины управления основными данными, которую мы обсудим в главе 9.



Принципы определения области видимости и разделения применимы также к бизнес-анализу и аналитике. Если, например, одна область предоставляет свои аналитические модели другой, они должны быть отделены с помощью шаблонов интеграции от уровня данных.

Разобравшись с вариантами, инструментами интеграции и объемом, можно приступать к расширению DDS дополнительными инструментами. Это нужно потому, что интеграция данных сама по себе не оказывает прямого влияния на бизнес. Истинная ценность данных зависит от знаний, новых идей и действий. Для описания, обобщения и демонстрации того, что произошло, лучше пользоваться инструментами бизнес-аналитики.

Возможности бизнес-аналитики

Инструменты бизнес-аналитики помогают предприятиям рассматривать данные более структурированно, обеспечивая при этом глубокую интерпретацию. Это позволяет принимать решения через интерактивный доступ и анализ данных. Для инструментов бизнес-аналитики, таких как отчеты и информационные панели, существует еще ряд соображений, которые помогут бизнесу извлечь дополнительные выгоды. Главный вопрос в том, нужно ли абстрагировать DDS с помощью *семантического слоя*, чтобы представить доступные данные пользователям в понятной форме для легкого и последовательного использования¹. Такой шаг дает ряд преимуществ.

- Семантические слои дают возможность пользователям легко запрашивать данные и создавать правильные комбинации. Обычно пользователям не нужно беспокоиться о соединениях данных. Отношения предопределены и помогают составить правильные комбинации, избегая ошибочных вычислений или интерпретаций. Поля имеют единообразные названия, форматируются и настраиваются. Например, поле суммы может суммироваться автоматически или поле данных может использоваться для расчетов, ориентированных на время.

¹ Семантический слой — это бизнес-представление данных, которое помогает конечным пользователям автономно получать доступ к данным с использованием общепринятых бизнес-терминов.

- Данные в семантических слоях оптимизированы (предварительно интегрированы, объединены и кэшированы) для потребления с целью повышения общей производительности. В результате пользовательский интерфейс становится намного лучше и быстрее.
- Базовое хранилище данных разделено. Это означает, что изменения в его структуре данных не обязательно сразу повлияют на все отчеты и информационные панели.
- Семантические слои могут иметь дополнительные функции, такие как управление версиями, безопасность на уровне строк и мониторинг.

Семантические слои имеют множество форм и часто тесно связаны со средой отчетности и информационными панелями. Tableau (<https://www.tableau.com/>), Qlik (<https://www.qlik.com/>), MicroStrategy (<https://www.microstrategy.com/>) и Microsoft Power BI (<http://powerbi.microsoft.com/>) — хорошо известные решения в этой области, обеспечивающие огромное количество функций. Еще один интересный вариант — AtScale (<https://www.atscale.com/>), который предоставляет только механизм оперативной аналитической обработки (online analytical processing, OLAP) и позволяет использовать любой инструмент, который вам нравится.

Большинство этих инструментов дают возможность создавать семантические слои двумя разными способами. Первый — использовать закрытую модель в памяти — модель кэширования. При выборе этой модели данные из базового хранилища передаются (обновляются) по расписанию на уровень кэширования. Это означает дублирование данных. Существует дополнительная задержка, потому что базовое хранилище может быть более актуальным, чем уровень кэширования.

Второй вариант — использовать базовое хранилище данных. В этой модели запросы передаются в базовый источник данных, поэтому семантический слой — это только уровень метаданных, который напрямую объединяет запросы. Такая модель лучше подходит для операционной отчетности и отчетности в режиме реального времени. Потому что БД обновляется за несколько секунд, без задержки. Главный недостаток в том, что данные должны быть смоделированы и оптимизированы для структуры отчетности, как правило кубической.

У семантических слоев есть недостатки, поэтому нужно пользоваться ими аккуратно. Для простых отчетов и предсказуемых запросов не обязательно добавлять издержки в виде дополнительного уровня. Другая проблема в том, что расширенные инструменты могут легко собирать и хранить большой объем данных. В конечном итоге это делает семантический слой дорогостоящим. Каждое решение для создания отчетов имеет свой семантический слой. Это может означать, что вы не раз будете переплачивать, ведь логика интеграции не может быть разделена между инструментами.

Сложность построения семантических слоев и решений для отчетности состоит в необходимости четко понимать, какие идеи вы хотите извлечь из данных. Необходимо заранее определить бизнес-вопросы, цели организации и источники данных. Во многих случаях лучше сначала оценить, изучить и обнаружить, какие действенные шаблоны можно извлечь из данных. Вот почему инструменты самообслуживания для обнаружения и подготовки данных становятся все более популярными.

Возможности самообслуживания

Возможности самообслуживания, такие как инструменты обнаружения, подготовки и визуализации данных, направлены на то, чтобы сделать процесс извлечения ценности из данных более быстрым за счет предоставления простых в использовании, интуитивно понятных функций самообслуживания для изучения, объединения, очистки, преобразования и визуализации данных. Термин «подготовка данных» указывает на то, что управление данными и предварительная обработка делают их более удобными для пользователя. Аналитики данных и специалисты могут выполнять эти действия самостоятельно, не владея сложными навыками программирования¹.

ОБНАРУЖЕНИЕ И АНАЛИЗ ДАННЫХ

Некоторые используют термины «обнаружение данных» и «анализ данных» как синонимы, но они означают не одно и то же. *Обнаружение данных* — это более широкий термин, используемый для описания итеративного процесса поиска закономерностей и тенденций в данных. Обычно это первый шаг в анализе данных, который можно сделать вручную или автоматически с помощью передовых методов, таких как машинное обучение. *Анализ данных* — это получение более глубокого понимания внутреннего содержимого данных и их характеристик.

Предоставление специалистам в области данных возможности превращать данные в ценность влияет на новую архитектуру. Так происходит потому что инструменты управления данными и конечными пользователями должны быть хорошо интегрированы и доступны. Специалисты по обработке данных разделяются по сегментам в зависимости от навыков и необходимых инструментов. Такое разделение и самообслуживание влияет на стандартизацию. Десятки инструментов сложно интегрировать и автоматизировать. Поэтому для успешного внедрения самообслуживания нужно сократить количество инструментов

¹ Джозеф Хеллерштейн (Joseph Hellerstein) написал интересную научную статью (<https://oreil.ly/G2X7z>) об обработчике данных.

бизнес-анализа и аналитики с дублирующими функциями, а также стараться использовать стандартизированные решения для конкретных нужд предприятия. Таблица 8.2 может помочь вам определить, какие инструменты подходят для конкретных специалистов в области данных.

Таблица 8.2. Функции бизнес-анализа и аналитики для специалистов по данным

	Аналитик данных	Специалист по данным	Инженер по обработке данных	Бизнес- пользователь
ETL/ELT: ETL означает «извлечение, загрузка и преобразование (extract, load and transform)». Эти шаги — необходимые функции приложения для перемещения данных из одного приложения в другое и преобразования их в правильную форму	Нет	Нет	Да	Нет
Обработка данных: функции приложения для очистки, упорядочения, объединения и преобразования данных для дальнейшего ETL	Да	Да	Да	Нет
Специальные запросы: функции приложения для произвольного создания форм анализа или отчетов. Анализ должен быть уникальным и конкретным	Да	Да	Нет	Нет
Семантическое моделирование и отчетность: функции приложения для создания бизнес-представлений данных. Функции приложения должны хранить данные в кубах OLAP (многомерных массивах данных)	Да	Нет	Да	Нет
Экспериментирование: функции приложения для проведения экспериментов по тестированию новых сценариев; приложения для проверки гипотез	Нет	Да	Нет	Нет
Визуализация и информационные панели: функции приложения для разработки привлекательных визуальных элементов и представлений	Нет	Да	Нет	Да
Совместное использование и распространение: функции приложения для обмена и передачи информации другим отделам	Да	Нет	Нет	Да

Эти инструменты имеют прямое отношение к управлению данными. Я вижу четкое различие в архитектуре между самообслуживанием и управляемыми данными.

- *Управляемые данные* (рис. 8.7, *вверху*) нужны для удовлетворения текущих потребностей бизнеса и поэтому по своей природе стабильны, автоматизированы и стандартизированы. Они тесно связаны с управлением метаданными и процессами и должны быть повторяемыми и эффективными. Предполагается, что решения самообслуживания, когда пользователи начинают с большой базы и в последний момент отбрасывают внушительное количество результатов, будут преобразованы в эффективные конвейеры данных при перемещении в управляемую среду.
- *Данные самообслуживания* (см. рис. 8.7, *внизу*) предназначены для специального и однократного анализа. Поэтому они являются временными по своей природе. Они начинаются с попытки понять, что находится в данных и какие существуют возможные отношения. Анализируя, комбинируя и корректируя вручную, вы можете проверить общую гипотезу. Эти действия могут также включать эксперименты с данными. Как только результаты ваших действий по обнаружению и исследованию станут ясными, можно осваивать ценность, что означает создание правильно управляемого конвейера данных (модель и ETL) в управляемой среде с реальными данными. Самообслуживание также можно использовать, чтобы помочь бизнес-пользователям быстро ответить на один конкретный бизнес-вопрос. Это называется *специальным анализом*.



Рис. 8.7. Современная архитектура данных предоставляет прикладные функции для действий с данными самообслуживания и управляемых действий

Существует четкое различие между самообслуживаемыми и управляемыми данными: у них разные принципы проектирования. Поэтому всегда должна осуществляться надлежащая передача. Для действий, связанных с самообслуживанием, вы можете рассмотреть структуры папок проекта, используя расположения ввода и вывода. Возможности самообслуживания предлагаются в виде платформы, данные в этой среде должны быть временными, и их можно сохранить только при соблюдении строгих условий.

Управляемые данные можно выгружать в среду самообслуживания. Обратный процесс (самообслуживание данных в управляемых средах) запрещен, потому что управляемые данные никогда не должны зависеть от вмешательства человека. Переход от самообслуживания к управлению должен быть плавным. Пользователи, обслуживающие ИТ, должны тесно сотрудничать с теми, кто использует среду самообслуживания. Инженер, например, может изучить задачи самообслуживания и даже помочь создать несколько образцов сценариев ETL, которые сначала тестируются в среде самообслуживания, а затем попадают в управляемую среду.

Для сред самообслуживания важно объединить все данные. Если данные фрагментированы и разрознены, пользователям будет сложно их быстро получить и объединить. Поэтому стоит воспользоваться интеллектуальным распределением для предоставления доступа всем пользователям. Современный подход к объединению и распределению данных — использование облачных инструментов, которые автоматически масштабируют задачи с помощью планирования и обрабатывают воспроизведение и безопасность.

Итеративный процесс самообслуживания и преобразования данных в интегрированные данные можно использовать как для бизнес-анализа, так и для аналитики, что подводит нас к следующему разделу.

Возможности аналитики

Продвинутая аналитика, как описано в главе 1, сосредоточена на прогнозировании поведения, событий и будущих тенденций. Это самая сложная форма создания ценности, ведь для нее нужны технологичные статистические модели с машинным обучением или искусственным интеллектом. Хотя обучение и разработка точных моделей становится все проще, их внедрение в производство, особенно в больших масштабах, является серьезной проблемой¹. Давайте разберем причины.

¹ Этот документ Google (<https://oreil.ly/6MaH->) описывает скрытый технический долг в системах машинного обучения. Обычно только небольшая часть реальных систем машинного обучения состоит из кода. Нужная окружающая инфраструктура часто бывает громоздкой и сложной.

- Модели сильно зависят от конвейеров данных. Использовать автономные и подготовленные данные легко, но на производстве все должно быть автоматизировано, качество данных должно быть гарантировано, модели должны автоматически переобучаться и развертываться, а для проверки точности после использования свежих данных требуется одобрение человека.
- Многие модели обычно создаются в изолированной экспериментальной среде для анализа данных без учета масштабируемости. Различные фреймворки, языки, библиотеки из интернета и собственный код часто смешиваются и объединяются. При организационной структуре с разными командами и без надлежащей «передачи» сложно интегрировать все в производство.
- Управление моделями в производстве — это другая история, потому что в производстве все должно постоянно контролироваться, оцениваться и проверяться. При принятии решений в режиме реального времени необходимо гарантировать эффективность и точность оценки. Вы же не хотите оказаться в ситуации, когда через два дня вы узнаете, что все ваши клиенты получили бесплатную рекламу.

Помня об этих проблемах, я хочу создать эталонную архитектуру, учитывая ряд основных принципов. Они поддерживаются рядом компонентов, чтобы сделать архитектуру управляемой.

Стандартная инфраструктура для автоматизированного развертывания

Первый принцип — использовать предварительно настроенные и изолированные платформы для экспериментов с данными, разработки и тестирования моделей. Они должны быть точно такими же, как и производственные платформы. Хотя вы можете использовать виртуальные машины, сегодня наиболее популярный выбор — это контейнеры. *Контейнеры* — это стандартные единицы программного обеспечения, которые объединяют фреймворки, библиотеки, зависимости, код, компиляторы — все, что необходимо для быстрой и надежной работы модели или приложения. Большое преимущество в том, что контейнер будет вести себя одинаково при разработке, тестировании и промышленной эксплуатации. Кроме того, нужна система или среда для оркестрации и управления всеми этими различными контейнерами. *Kubernetes* — один из наиболее очевидных вариантов, поскольку поддерживается всеми крупными поставщиками облачных вычислений и платформ. Эта платформа заботится о развертывании, подключении к сети, изоляции и планировании всех ваших контейнеров.

Вы можете выбрать бессерверные модели как альтернативу. Популярные поставщики облачных сервисов предлагают услуги с небольшим количеством кода/без кода, позволяющие использовать машинное обучение и не сильно заботиться об обслуживании платформы.

Модели без сохранения состояния

Следующий принцип — платформы выполнения моделей не имеют состояния. Они не сохраняют и не хранят данные, хотя могут создавать временные или хранить справочные данные. Любые данные, которые они должны использовать, будут поступать из выходной папки и записываться в нее. Это важно, так как означает, что вы можете легко заменять модели без перемещения или миграции данных. Этот принцип применим ко всем модельным схемам обслуживания, что подводит нас к следующему принципу.

Предварительно настроенные рабочие места

Вместо того чтобы заставлять специалистов по данным тратить много времени и усилий на настройку сред, я рекомендую разрабатывать, стандартизировать и устанавливать принципы использования инструментальных средств для обработки данных. Например, предварительно настроенное ПО со стандартными технологиями, такими как языки и библиотеки, позволяет специалистам по данным работать эффективно. Некоторые крупные предприятия, такие как Uber, добиваются успеха, используя развернутые версии Jupyter Server (<https://jupyter.org/>), VSCode Server (<https://oreil.ly/LCAd1>) и RStudio Server (<https://oreil.ly/65gQk>)¹. Предоставление специалистам по обработке данных прямого доступа к этим инструментам с предварительно настроенными папками проекта для кода, входных и выходных данных действительно поможет ускорить их работу.

Для масштабируемости я рекомендую стандартизировать языки, фреймворки и шаблоны интеграции. Если каждому специалисту по данным разрешить выбирать собственный язык и версию, общее обслуживание архитектуры станет кошмаром, поэтому желательно ограничить количество языков. Далее я рекомендую стандартизировать выбранные фреймворки; вы можете использовать Spark, Flink, PyTorch, Scikit, TensorFlow или MLflow.

¹ Uber опубликовал руководство (<https://eng.uber.com/dsw>) по анализу турбокомпрессора с помощью инструментов для анализа данных.

Чтобы сделать рабочие места более надежными, их можно дополнить автоматизированными процедурами сбора метаданных, файлов журналов, состояния модели и т. д. Рассмотрите также возможность предоставления всем командам повторно используемого кода (фрагментов), чтобы упростить, ускорить и согласовать разработку моделей. Объединение всех дисциплин в одно стратегическое подразделение, например, центр передовых технологий может помочь упростить все аналитические усилия вашей организации.

Стандартизация шаблонов интеграции моделей

Следующий шаг — признать, что есть разные шаблоны интеграции, которые можно стандартизировать. Рассмотрим несколько шаблонов интеграции рабочих сред в производство.

- *Модель как пакетный ввод/вывод.* В этом подходе модель принимает, обрабатывает и выдает данные пакетами или мини-пакетами. Фрагменты, подмножества данных или данные целиком с метками будут использоваться для обучения, прогнозирования и выработки рекомендаций. Пакеты обычно представлены наборами файлов, например в формате CSV.
- *Модель как поток.* В этом подходе модель реактивно взаимодействует с потоком данных, где части данных прибывают последовательно, друг за другом. Если требуются дополнительные данные, модели можно разрешить читать хранилище DDS напрямую. Эта модель может генерировать и публиковать новые события.
- *Модель как API.* Модель этого вида развертывается как веб-служба, поэтому может использоваться другими приложениями и процессами. Чтобы вызвать модель и получить прогноз, необходимо выполнить вызов API.

В зависимости от желаемой степени стандартизации процессов разные подходы можно связать с разными инструментальными средствами. Например, обработка больших объемов данных в этой модели может выполняться с помощью Spark. Используя и интегрируя модели других команд разработчиков, помните, что независимо от применяемого шаблона нужно заключить договор о совместном использовании данных (см. подраздел «Контракты на поставку данных и соглашения о совместном их использовании» на с. 63).

Автоматизация

Для продвинутой аналитики необходим хорошо спроектированный конвейер данных, поэтому большая часть вашего внимания должна быть сосредоточена на автоматизации. Это тоже непростая работа. Чтобы добиться успеха, нужно

все связать вместе. Для организации шагов конвейера данных я настоятельно рекомендую Apache AirFlow. Для непрерывной доставки обратите внимание на GoCD (<https://www.gocd.org/>), а для непрерывной интеграции — на Jenkins (<https://jenkins.io/>), CircleCI (<https://circleci.com/>) или Bamboo (<https://oreil.ly/qyZQo>). Для репозиториев исходного кода наиболее популярным вариантом является любой из фреймворков на основе Git (<https://git-scm.com>). Вы можете положиться на поставщиков облачных платформ или сервисов. Крупные компании предлагают машинное обучение как услугу (Machine Learning as a Service, MLaaS), интегрированные среды, которые очень хорошо подходят как для разработки, так и для внедрения в производство. Databricks (<https://databricks.com/>), Qubole (<https://www.qubole.com/>), Azure Machine Learning (<https://oreil.ly/XRyyK>), AWS SageMaker (https://oreil.ly/P_O3w) и Google AI Platform (<https://oreil.ly/AE5qx>) — популярные варианты.

Метаданные модели

Последняя область, на которую нужно обратить внимание при установлении принципов, — это создание моделей с метаданными. Возможно, вы захотите применить управление версиями, чтобы отслеживать, какие данные использовались для обучения с помощью разных моделей. DVC (<https://dvc.org/>) — популярная система контроля версий с открытым исходным кодом для проектов машинного обучения. Чтобы управлять версиями самой модели, рассмотрите возможность сериализации¹. Сохраните модель как версию и управляйте ею с использованием той же инфраструктуры, которую вы применяли для управления версиями данных. Рассмотрите возможность хранения всех этих метаданных в центральном репозитории кода.

Вместо управления версиями данных вы можете создавать версии конвейеров данных. Как описано в главе 3, все данные, хранящиеся в RDS, неизменяемые, что позволяет воссоздавать те же самые данные снова и снова. Но вы должны быть уверены, что результаты всех конвейеров данных будут одинаковыми, поэтому управление версиями конвейеров — важный пункт.

Вы также должны собирать информацию о фреймворках моделирования, контейнерах и методах моделирования для каждой модели. Например, в каких моделях используются методы Монте-Карло (<https://oreil.ly/SMY6R>) и «случайного леса» (<https://oreil.ly/SDC3o>)? Здесь вы можете добавить классификаторы, если модель полностью прозрачна в том, как она принимает решения, или действует

¹ Pickle (<https://oreil.ly/zK5Xq>) — популярный инструмент сериализации объектов в сообществе Python.

больше как черный ящик. Регулирующие органы могут задавать вопросы о том, какие модели используются, и вы должны быть к этому готовы.



При использовании некоторых методов результаты невозможно воспроизвести. Входные данные и метод могут быть одинаковыми, но из-за случайной составляющей в расчетах результаты всегда будут отличаться!

Вы можете добавить независимые от модели форматы обмена для распространения или совместного использования моделей. Например, язык разметки прогнозных моделей (<https://oreil.ly/pvAgV>) — это стандарт XML для обмена модельями между средами. Наконец, определите, какие признаки и метки использовались.



Признаки — это столбцы во входных данных. Например, если вы пытаетесь предсказать чей-то возраст, вашими входными характеристиками могут быть рост и цвет волос. *Метка* — это окончательный результат: в данном случае 12, 78 и т. д.

Чтобы упростить работу групп, вы можете предложить заранее подготовленные контейнеры для удобного ввода и вывода данных с заранее заданными местоположениями файлов, которые автоматически регистрируют метаданные и т. д. Этот способ работы аналогичен тому, о котором вы читали во врезке «Микросервисы и метаданные» на с. 149.

Эталонная архитектура продвинутой аналитики

Давайте соединим все точки, чтобы создать последовательный, повторяемый и надежный процесс сборки и автоматического развертывания аналитических моделей. Архитектура, как вы поняли, должна поддерживать конвейеры как пакетных, так и потоковых данных и содержать возможности, необходимые для разработки моделей. Наконец, она должна предлагать инструменты для перемещения модели по этапам разработки и выпуска, поддерживаемые функциями ведения журналов, мониторинга и метаданных для наблюдения за всем. Когда все будет собрано вместе, мы получим результат, который показан на рис. 8.8.

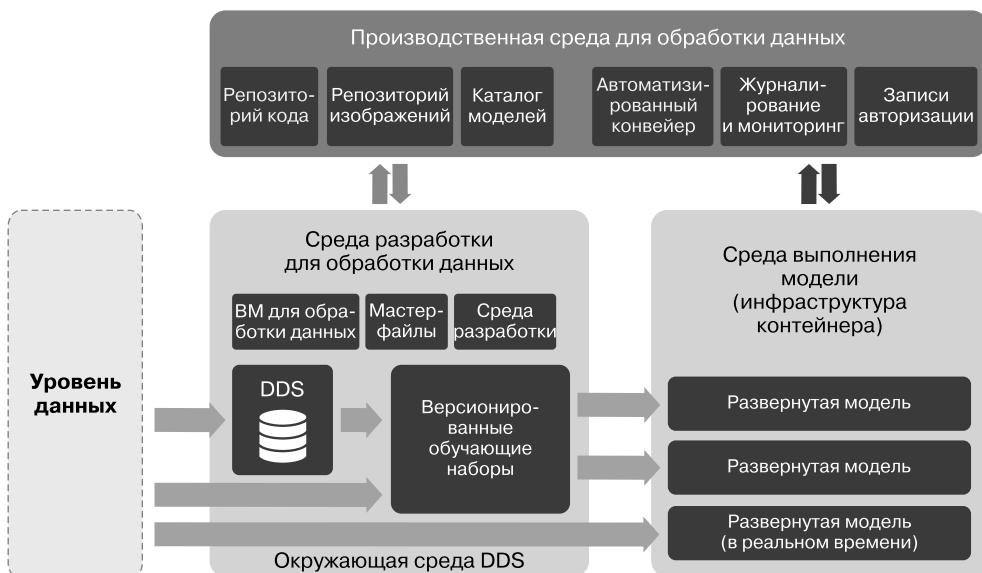


Рис. 8.8. Эталонная архитектура платформы продвинутой аналитики для создания, обучения и развертывания аналитических моделей

Такая эталонная архитектура — лишь пример того, как можно построить автоматизированный процесс разработки модели и управления ею. В этой модели я сделал разработки и версионированные обучающие наборы частью среды DDS потребителя данных. Это связано с тем, что требования и контекст у разных потребителей различаются. Вспомогательные возможности, такие как репозитории изображений и каталоги, расположены централизованно для более строгого управления и контроля. Например, предварительно настроенные изображения нужно предоставлять централизованно, потому что для обеспечения масштабируемости мы должны стандартизировать наиболее часто используемые платформы и языки. Пара моментов, которые могут вызвать изменения в этой эталонной архитектуре.

- Конвейеры могут быть спроектированы так, чтобы требовать утверждения вручную после повторного обучения или автоматически останавливать работу после того, как качество данных достигает определенных пороговых значений.
- Повторное обучение модели может инициироваться несколькими событиями: выпуск обновленной модели, бизнес-событие, событие поступления новых данных, генерированное вручную, и т. д. Конвейер развертывания модели может быть более тесно связан с конвейером разработки данных. Например, модель автоматически сохраняется и развертывается при поступлении новых данных.

- Проект и структура DDS могут варьироваться в зависимости от требований. Возможно, вы захотите использовать проект многовариантной БД для проверки различных структур данных и чтения шаблонов.
- Чтобы отслеживать систематическую погрешность, достоверность и качество моделей, вы можете фиксировать выходные данные, автоматически исследовать и сравнивать результаты конкретных значений характеристик с общей совокупностью. В этой ситуации могут работать одновременно несколько моделей.
- Для объяснения может потребоваться автоматическое создание версий и методов всех обучающих данных и моделей.
- Есть много способов для возвращения выходных данных моделей, включая асинхронное выполнение и вычисление пакета путем добавления новых записей данных, обслуживания небольших строк через веб-службу запроса/ответа или создания новых событий путем анализа других.
- Неструктурированные данные могут изменить архитектуру. Возможно, вы захотите использовать внешние или виртуальные сервисы или делиться данными с внешними сторонами.

Как вы понимаете, такие действия сильно зависят от бизнес-требований моделей и от того, как эти модели используются или интегрируются в другие последующие процессы. Помимо этих вариантов, эталонная архитектура с ее автоматизацией, управлением версиями, неизменяемыми данными и предварительно настроенными артефактами поможет сделать аналитику и разработку моделей более масштабируемыми.

Давайте обобщим. Схема на рис. 8.9 может показаться громоздкой, поэтому я оставил ее напоследок. Возможности бизнес-анализа и аналитики — важная часть общей архитектуры. Они позволяют бизнесу превращать данные в ценность и работать в тесном сотрудничестве. При правильной разработке — в виде модели сервиса — они сделают доставку ценности масштабируемой. Ожидается, что все эти возможности будут тесно взаимодействовать с уровнем данных.

На рис. 8.9 показана другая точка зрения: взгляд в более широком смысле от поставщиков до потребителей данных. DDS на этом рисунке расположены справа, потому что они облегчают создание новых данных. Это подойдет только для сценариев, требующих создания сложной бизнес-логики, или сценариев с большими объемами данных, которые нужно предварительно обработать и сохранить в другом месте. Для сценариев, которые не требуют создания новых данных, ожидается, что инструменты потребления будут использовать RDS напрямую.

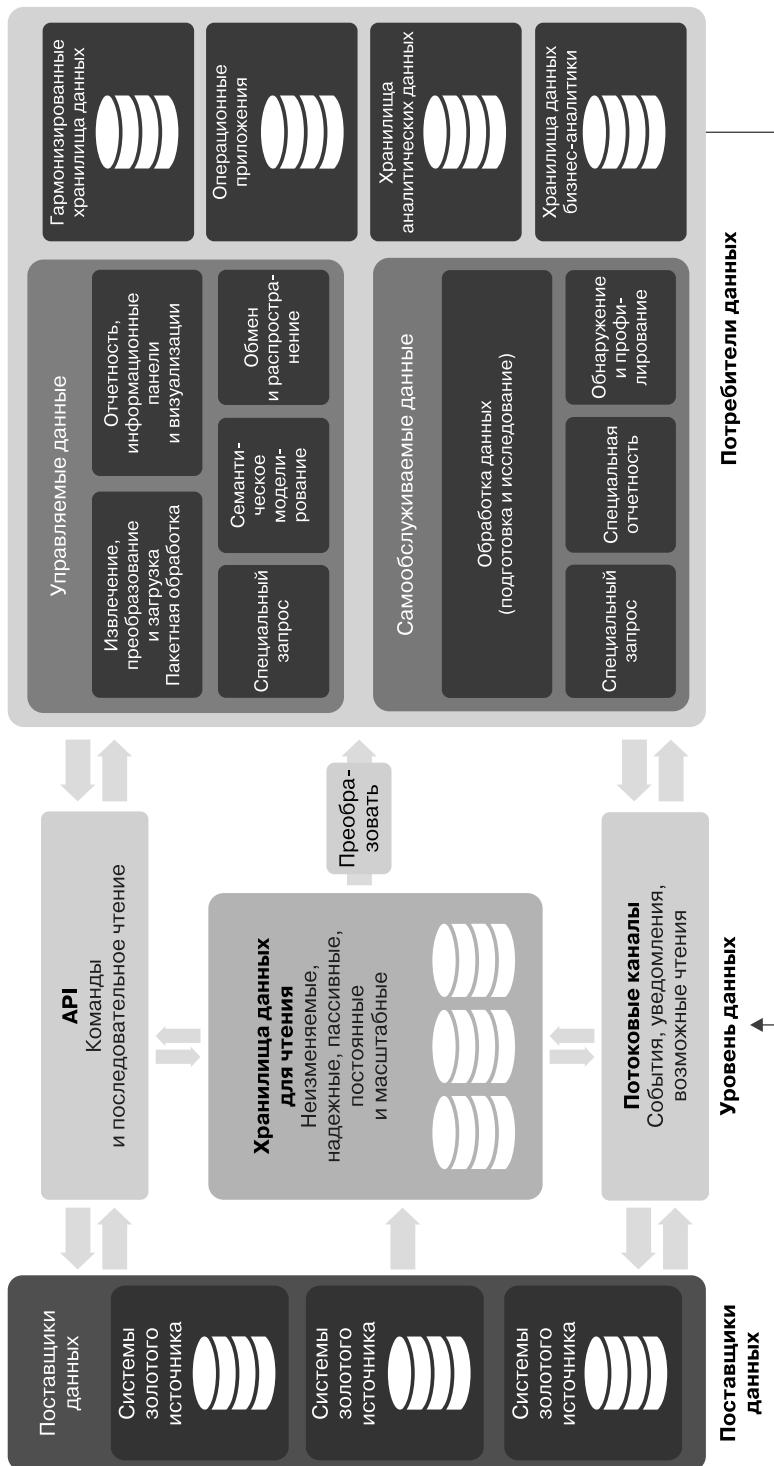


Рис. 8.9. Общая эталонная архитектура, показывающая различные типы и шаблоны использования DDS

Итоги главы

Архитектура, которую мы создавали на протяжении всей этой главы, помогает управлять производством, продвинутой и бизнес-аналитикой, такой как машинное обучение, в любом масштабе. Вы видели, что, серьезно относясь к управлению данными и автоматизации, мы демократизируем данные в целом. Мы коснулись самообслуживания и действий с управляемыми данными, а также принципов, которые их поддерживают. Еще вы узнали, что для быстрой окупаемости важно отказаться от ручных действий, таких как управление версиями, мониторинг аналитики и развертывание. Это требует другой культуры, выходящей далеко за рамки традиционного моделирования данных. Чтобы овладеть этими навыками, ваши инженеры по обработке данных должны стать хорошо обученными или опытными архитекторами ПО. Для этого потребуется широкое распространение самообслуживания.

Наконец, вы узнали, как устраниТЬ разрозненность и сводить к минимуму зависимости между DDS. Разделение и изоляция помогают командам оставаться сосредоточенными. Вы также узнали о принципах совместного использования общих данных в области между подобластями. Этот шаблон является мостом к следующей главе, в которой мы обсудим управление основными и справочными данными.

ГЛАВА 9

Управление основными корпоративными данными

В предыдущих главах вы узнали о масштабе, в котором предприятиям нужно управлять данными и распределять их. Предприятия обычно используют большое количество областей, каждую со своими системами и данными. Это сложно, ведь данные распределены по всему миру и может существовать несколько версий одних данных. Интеграция, например, обеспечивает всестороннее изучение ваших клиентов и требует больше усилий, так как она нуждается в объединении и согласовании всех независимых частей одних и тех же данных из разных областей. Другая проблема в том, что данные могут быть несовместимыми в контекстах между различными областями и могут быть различия в уровнях качества данных.

Чтобы решить эти проблемы, нам нужна дисциплина *управления основными данными* (*мастер-данными*) (master data management, MDM). Дисциплина MDM, как описано в главе 1, предназначена для управления и распределения критически важных данных, чтобы обеспечить согласованность, качество и надежность. Это важно, потому что непоследовательные и неверные данные могут подорвать доверие, а также снизить доходы и прибыль. Другие тенденции, стимулирующие спрос на управление мастер-данными, — это безопасность, обнаружение мошенничества и нормативные акты, такие как европейский Общий регламент по защите данных (General Data Protection Regulation, GDPR) и Закон штата Калифорния о защите конфиденциальности потребителей (California Consumer Privacy Act, CCPA). Неэффективность в управлении мастер-данными может привести к неспособности обнаружить мошенничество или к штрафам со стороны регулирующих органов.



Дисциплина MDM привносит больше взаимосвязей в архитектуру. Она улучшает согласованность данных предприятия, но также увеличивает связанность архитектуры. Чем больше данных вы контролируете, тем более связанной будет ваша архитектура.

Прежде чем показать, как применять MDM в большой экосистеме, я быстро резюмирую, что такое MDM, исследуя четыре стиля реализации. После чего мы рассмотрим, в каком масштабе можно применять MDM. Наконец, мы обсудим принципы предоставления областям возможности самостоятельного распределения гармонизированных данных.

Демистификация управления мастер-данными

В большинстве организаций есть системы, использующие общие данных. Возьмем для примера такие понятия, как «клиент» и «товар»: почти наверняка разные версии одних и тех же мастер-данных хранятся в разных частях организации, во многих приложениях и системах. Такое дублирование не только неэффективно, но и может привести к несогласованности. Имя клиента может быть написано по-другому, адреса могут отличаться, у каждого покупателя вполне может быть свой уникальный идентификационный номер; каждая копия данных может принадлежать разным отделам или областям. Если данные о клиентах изменяются или их необходимо объединить, как вы узнаете, кем являются одни и те же клиенты, несмотря на эти различия? Именно здесь на помощь приходит MDM. Эта дисциплина фокусируется на обнаружении и устраниении несоответствий данных путем распределения основных версий данных между приложениями и системами.

Стили управления основными данными

Для успешного управления основными и справочными данными вам необходимо разработать и внедрить решение MDM. Лин Робинсон (Lyn Robison) (<https://oreil.ly/Pki76>), пишущий для Gartner, выделяет четыре стиля управления MDM: консолидацию, реестр, централизацию и сосуществование (рис. 9.1)¹.

- *Стиль консолидации.* Стиль консолидации во многом перекликается с хранилищем данных, поскольку он объединяет основные данные с остальными

¹ Robinson L. A Comparison of Master Data Management Implementation Styles. 6 ноября 2017 г. <https://oreil.ly/F3o4A>.

в единый репозиторий или ядро, называемое *хранилищем мастер-данных* или *ядром MDM*. Это ядро объединяет базу данных с программным обеспечением для непрерывного сбора мастер-данных из операционных систем для повышения качества, управления и обеспечения синхронизации мастер-данных с другими системами. Данные становятся доступными из этого объединенного репозитория и могут использоваться несколькими приложениями. Этот стиль обычно применяется для анализа, бизнес-аналитики (business intelligence, BI) и отчетности. Не прилагается никаких усилий для очистки или улучшения данных в системах с золотыми источниками. Улучшения, внесенные в данные, ограничиваются ядром, поэтому их преимуществами пользуются только потребляющие приложения.

- *Стиль реестра.* Самый простой для реализации стиль MDM – стиль *реестра*. Реестр – это простая таблица перекрестных ссылок, дающая непосредственное представление об объектах, которые были проверены на наличие дубликатов, и о том, какие отношения были обнаружены между ними. Он использует внутреннюю справочную систему и локальные идентификаторы (например, идентификаторы клиентов) для обратной связи с исходными источниками данных.

Этот стиль не влияет на приложения с золотыми источниками. Каждая система-источник сохраняет контроль над своими данными и остается золотым источником. Данные из реестра не передаются обратно в системы-источники, поэтому отсутствует прямое влияние на качество данных. Хотя реестр – это самый простой способ реализации MDM, его функционал ограничен. Он предоставляет ссылки только между соответствующими записями.

- *Стиль централизации.* В стиле централизации хранилище основных данных становится централизованным репозиторием для всех основных данных из всех операционных и аналитических систем. Для этого нужно, чтобы все приложения и системы каждый раз получали и использовали основные данные из этого центрального репозитория, не задействуя свои собственные. Улучшенные данные должны немедленно передаваться обратно в соответствующие системы-источники.

Наибольшее беспокойство в этом стиле вызывает требование вторжения в системы с золотыми источниками для двусторонней синхронизации. Приложения должны соответствовать требованиям, чтобы использовать репозиторий основных данных в качестве «серверной части». Этот шаблон соответствия может быть трудно реализовать, потому что в проект приложения не всегда можно внести корректизы. Например, готовые продукты, как правило, не подлежат изменениям. Еще одна проблема – тесная связь: если потребуется изменить централизованный репозиторий по какой-либо

причине, придется изменить все другие системы. Коммерческие поставщики MDM скажут вам, что это лучший подход для хорошо управляемых и регулируемых основных данных, однако я советую серьезно рассмотреть другие подходы.

Стиль централизации обычно используется в ситуациях, когда важно централизованное управление. Классификации безопасности, организационные структуры и внутренняя информация о сотрудниках — это примеры данных, которыми можно управлять в стиле централизации.

- *Стиль сосуществования.* Стиль сосуществования применяется, когда основные данные не могут использоваться централизованно и должны распределяться в несколько мест по всему предприятию. В этом стиле улучшения возвращаются в системы-источники, поэтому обновления выполняются в системах с золотыми источниками. Сосуществование — наиболее сложный стиль для реализации. Для распространения и обеспечения согласованности данных между системами золотых источников и хранилищем основных данных нужно настроить сложные шаблоны интеграции данных. Конвейеры данных, необходимые для этой модели, могут варьироваться от систем реального времени до пакетов или API. Реализация может охватывать несколько сред, например локальную и облачную.

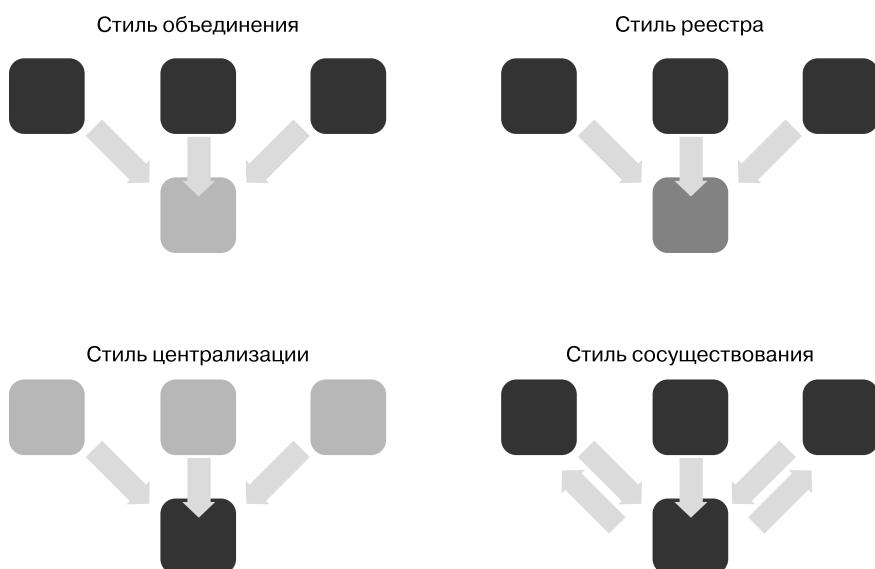


Рис. 9.1. Сравнение четырех стилей реализации управления мастер-данными от Gartner

Эти четыре стиля реализации не исключают друг друга. Вы можете начать, например, со стиля реестра, а после перейти к стилю сосуществования. Вы можете выбирать разные стили для разных наборов данных. Например, для структуры отчетности можно выбрать стиль консолидации, а для данных о клиентах — стиль сосуществования. При применении стиля вы можете смешивать разные реализации продукта. Важно отметить, что выбор стиля не должен зависеть от технологий. Организационные потребности и потребности данных — вот критерии выбора стиля.

Эталонная архитектура MDM

Интеграция данных — основа поддержки MDM. Многие реализации MDM должны поддерживать как пакетную обработку, так и обработку в реальном времени. В транзакционных системах обработка обычно происходит в режиме реального времени, а аналитические системы обычно фокусируются на пакетах большого объема. Дисциплина MDM должна опираться на ту же основу, которую мы использовали до сих пор для распределения и интеграции всех данных. В этой модели хранилище основных данных одновременно является потребителем и поставщиком данных.

Важно не устанавливать двухточечные соединения между ядром MDM и поставщиками или потребителями приложения. Иначе увеличивается риск рас согласования данных. Например, потребители могут получать новые записи, доставляемые напрямую из ядра MDM, в то время как уровень данных еще не обновился. Если ядро MDM всегда использует уровень данных, таких несответствий не будет.

Давайте посмотрим на рис. 9.2, чтобы увидеть, как работают распределение данных и объединение в MDM.

Процесс MDM начинается с определения того, какие приложения и системы и какие уникальные и заслуживающие доверия данные создают и поддерживают. Золотые наборы данных из систем золотых источников — основа для этого. Из-за того что в каждом приложении данные хранятся по-разному, важно понимать контекст и то, какие бизнес-правила, форматы и диапазоны ссылок влияют на основные данные в нем. Определив, какие данные дублируются, можно приступать к поиску источника данных (шаг 1 на рис. 9.2). Внедрение данных тесно связано с управлением данными. Во время этого процесса также должны доставляться метаданные, а владельцы данных должны тщательно описывать все атрибуты.

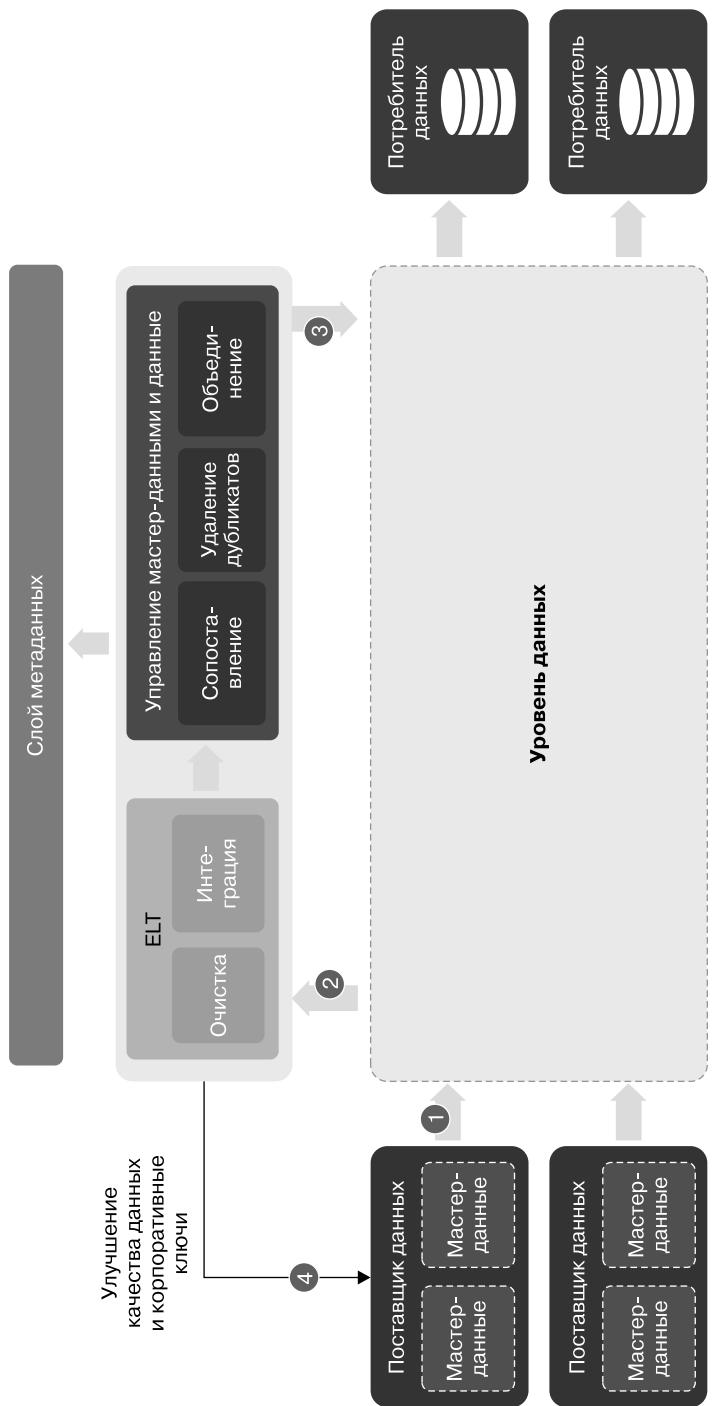


Рис. 9.2. Очень важно последовательно представлять основные данные в распределенной архитектуре. Поэтому все возможности управления основными данными должны использовать одни и те же архитектуры интеграции для обмена данными

Разработка решения для управления основными данными

Следующий шаг — разработка решения MDM, которое гармонизирует данные и делает их согласованными. Вы можете выбрать продукт от поставщика или спроектировать его самостоятельно, выполнив декомпозицию или разбивку отдельных функций приложения, которые предоставляет MDM. Популярные поставщики в области MDM — Orchestra (<https://oreil.ly/RkXSt>), SAP MDG (<https://oreil.ly/KdupQ>), Informatica Identity Resolution (<https://oreil.ly/yA4qx>), Tamr (<https://www.tamr.com/>), Dell Boomi (<https://www.boomi.com/>), Microsoft’s Master Data Services (SQL Server) (<https://oreil.ly/i0ylI>) и IBM Entity Analytics (<https://oreil.ly/MKbfX>).

Для ядра MDM вам может понадобиться реализовать технологии и шаблоны проектирования. Базы данных NoSQL, например, могут постоянно вводить вновь созданные данные и отслеживать глобальный идентификационный номер, прикрепленный к каждой записи. Источники событий (см. подраздел «Источники событий и команд» на с. 169) могут решить многие из проблем согласованности данных, которые обычно встречаются в распределенной архитектуре. Машинное обучение может улучшить качество и предсказать, какие данные принадлежат друг другу и какие глобальные идентификационные номера должны быть им присвоены.

Проектирование и реализация MDM (шаг 2 на рис. 9.2) тесно связаны с моделированием и интеграцией данных, потому что выбор перекрывающихся атрибутов данных, гармонизация (ETL) и преобразование их в подходящие модели данных — важная часть общего проектного решения MDM. В зависимости от стиля и выбранных решений объем работы по моделированию и интеграции данных может меняться, как и шаблоны интеграции и архитектуры. Например, стиль сосуществования больше полагается на обработку в реальном времени и использует архитектуры API и потоковой передачи, а для стиля консолидации идеально подходит архитектура RDS.

После объединения, согласования и интегрирования всех перекрывающихся данных в решение MDM важно настроить управление. Согласуйте с владельцами данных следующие пункты.

- Правила очистки, сопоставления, объединения и связывания.
- Для каких целевых классификаций (см. подраздел «Классификация целей» на стр. 238) центральное хранилище мастер-данных может использоваться как авторитетный источник и в каких ситуациях потребители данных должны использовать свои локальные (исходные) источники.
- Какие процессы качества данных нужно внедрить.
- Как быстро улучшения должны передаваться обратно в системы-источники.

- Кому принадлежат только что созданные данные.
- Методы интеграции данных для получения улучшений (ожидайте вариации и комбинации пакетов, API и потоковой передачи).
- Какие пользователи имеют право утверждать или отклонять предлагаемые изменения.
- Определения основных данных, которые должны храниться в центральных репозиториях метаданных, таких как каталог данных.

После того как вы успешно внедрили MDM и выполнили все вышеперечисленные шаги, вы можете выработать стратегию распределения данных обратно в системы-источники и другие приложения (шаг 3 на рис. 9.2).

Распространение MDM

При распространении данных MDM важно соблюдать баланс между согласованностью данных и низкой задержкой. Все распределенные архитектуры подчиняются теореме CAP (<https://oreil.ly/Tr2Tf>), а это значит, что строгая согласованность и низкая задержка не всегда могут находиться в балансе. В зависимости от вашей ситуации вам придется выбирать, чему отдать предпочтение.

Если варианты использования операционные или имеют небольшую задержку, применяйте архитектуру API. В случаях, когда возможна высокая задержка, мастер-данные можно распространять с помощью архитектур потоковой передачи и RDS. Для микросервисов подойдут хранилища состояний (см. врезку «Хранилища состояний» на с. 171). Вместо того перемещения миллионов записей одновременно вы предоставляете потребляющим приложениям, использующим потоковую архитектуру, результаты MDM в виде потоковой базы данных. Таким образом, микросервисы области могут представлять собой конгломераты сервисов из предметной области и микросервисов MDM, отвечающие за данные MDM.

Основные идентифицирующие номера

*Основные идентифицирующие номера*¹ — важный аспект MDM. Они связывают вместе полученные данные и данные из локальных систем. Такие элементы данных важны для определения, какие данные являются основными и как

¹ Основной идентификационный номер или основной идентификатор — это один из расширенных атрибутов, созданных MDM. Он гарантирует, что записи уникально идентифицированы, назначая уникальные идентификаторы и, что наиболее важно, документируя отношения между совпадающими записями.

они связаны друг с другом. Идентификация уникальных данных и назначение основных идентификаторов могут выполняться только глобально, а не локально внутри систем. Этот процесс требует собрать вместе все данные из разных систем. Это также означает, что в случае изменения данных в системах эти системы должны повторно отправить данные, чтобы снова запустить обнаружение.

Что касается золотых источников, то, как только они становятся предметом MDM, основные идентифицирующие номера должны быть переданы назад в системы-источники и сохранены там (шаг 4 на рис. 9.2). В качестве альтернативы предметные области могут поддерживать отношения между локальными и основными идентификаторами в таблице поиска. Эта таблица, описывающая, какие локальные данные каким основным данным принадлежат, может быть опубликована в решении MDM. Выявление и поддержка взаимосвязей важны для знания не только того, какие данные являются основными, но и какие данные можно быстро связать с другими данными. Если локальные (принадлежащие области) ключи в операционной системе изменятся, единственным элементом, связывающим все вместе, останется основной идентификатор.

При распространении основных идентификаторов в обратном направлении не экстраполируйте основные идентификаторы MDM на каждую операцию. Это может создать проблемы несогласованности. Только операции, на которые распространяется управление основными данными, должны получать основной идентификатор из ядра MDM. Системы, не подпадающие под действие MDM, должны использовать свою локальную целостность.

КЛЮЧИ ПРИЛОЖЕНИЯ И СУРРОГАТНЫЕ КЛЮЧИ

У инженеров по обработке данных есть два варианта моделирования проекта и выбора уникальных идентификаторов. Можно использовать уникальный существующий бизнес-идентификатор или ключ приложения либо генерированный системой (суррогатный) ключ, который остается уникальным, никогда не меняется и не может быть изменен. Преимущество использования суррогатного ключа в том, что он не имеет семантического значения. MDM назначает суррогатный ключ каждой основной записи, чтобы помочь распознать группы основных данных.

Когда данные перераспределяются и основные идентификаторы находятся в исходной системе администрирования, должен соблюдаться принцип их доставки вместе с данными. Включение основных идентификаторов во время перераспределения данных упрощает определение того, какие данные принадлежат друг другу.

Справочные и основные данные

Хотя многие решения MDM могут управлять как справочными, так и основными данными, я рекомендую четко различать их. *Справочные данные* — это данные, используемые для определения, классификации, организации, группировки или категоризации других данных (или иерархий значений, таких как отношения между продуктом и географическими иерархиями). *Основные данные*, напротив, касаются основных понятий.

Что касается справочных данных, то масштабируемая архитектура не указывает явно, какой стиль реализации использовать. Справочные данные обычно управляются централизованно в виде таблиц или документов. Центральное решение MDM используется для обеспечения согласованности, гарантируя, что все соответствующие заинтересованные стороны активно участвуют в изменениях и обеспечивают соблюдение нормативных требований и управление. Валютный стандарт ISO — прекрасный тому пример. Ожидается, что многие финансовые системы будут его придерживаться. Когда список валют изменяется, решение MDM управляет этими изменениями и распределяет все справочные данные через архитектуры интеграции в системы-потребители, следя за тем, чтобы все они использовали один и тот же согласованный список валют.

Руководящий принцип для справочных данных: «обязательное использование основных идентификаторов при распределении данных через уровень данных». При наличии метаданных и средств контроля качества данных вы сможете гарантировать качество данных, используя одни и те же согласованные данные.

Определение области видимости корпоративных данных

Один из самых сложных вопросов в управлении основными данными — как определить организационную область видимости основных и справочных данных. Какими данными вы должны управлять на уровне предприятия, а какие можно распределить по областям?

С MDM легко попасть в ловушку унификации корпоративных данных: расширение области видимости и обработка слишком большого количества результатов приведет к резкому росту объемов данных, требующих интеграции, управления и координации. Решение — метаданные. Основываясь на информации о происхождении, моделях данных и соглашениях о совместном использовании, можно найти области пересечения и общие интересы предметных областей, а затем определить область видимости. Приведу пример: атрибут клиента,

который используется только в пределах одной области, не должен входить в сферу действия MDM. Но, так как атрибуты, распределенные между всеми областями, являются потенциальными кандидатами, любое решение MDM должно работать в тесном взаимодействии с поддержкой метаданных.

Позже, когда качество ваших метаданных достигнет высокого уровня, подумайте над использованием машинного обучения для поиска пересечений областей и выявления данных для управления на уровне предприятия. Модели могут узнать и предсказать, какие данные важны. Я еще не видел, чтобы компании делали это, но для меня это был бы высочайший уровень зрелости, которого может достичь компания.

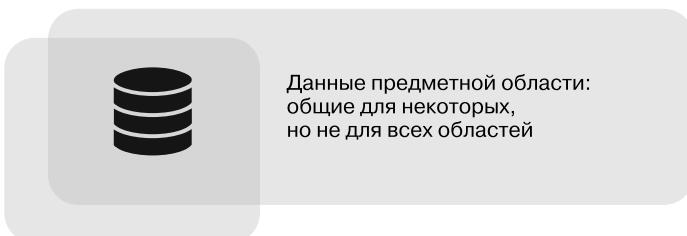
Во время поиска пересекающихся данных вы можете обнаружить различную степень пересечения. Одни данные — более общие и охватывают большое количество областей. Другие имеют ограниченное пересечение и охватывают лишь несколько областей. Чтобы различать важность и степень пересечения, я использую следующие классификации данных.

- *Корпоративные данные* (рис. 9.3) — это данные, применяемые в масштабе всего предприятия. Они содержат как основные, так и справочные данные. Их согласованность важна, и их область видимости может варьироваться от нескольких до всех областей или даже за пределами предприятия.



Рис. 9.3. Корпоративные данные — это данные, используемые на всем предприятии

- *Данные предметной области* (рис. 9.4) — это данные, которые совместно используются некоторыми, но не всеми областями. Они важны для управления пересекающимися областями, но не имеют глобального значения для всего предприятия, а также не имеют никакого нормативного значения. Данные предметной области обычно управляются и обслуживаются в отдельных областях, а не на глобальном уровне.
- *Локальные данные* (рис. 9.5) — это данные, которые не перемещаются и не используются другими областями. Они востребованы только в пределах одной локальной области или системы.



Данные предметной области:
общие для некоторых,
но не для всех областей

Рис. 9.4. Данные предметной области — это данные, совместно используемые несколькими областями



Локальные данные: используются
исключительно в границах одной
«локальной» предметной области

Рис. 9.5. Локальные данные — это данные, которые не покидают
границ предметной области

Эти классификации помогут вам реализовать управление основными данными и данными в целом. Корпоративные данные имеют высокий уровень согласованности и множество зависимостей. Отношения и согласованность более важны, потому что многие области и их данные зависят от корпоративных данных. Правильное управление этими данными важно и помогает реализовать дополнительные элементы управления интерфейсами, схемами метаданных и процессами распределения данных. Одна прагматическая форма контроля — отслеживать каждый золотой набор данных, содержащий основные или справочные данные, и его отношения с корпоративными данными. Эта информация может храниться в списке золотых источников. По мере прохождения данных через архитектуру вы можете использовать эти метаданные и проверять целостность основных идентификаторов с помощью функций качества данных.



Основные идентификаторы в корпоративных данных еще называются *корпоративными идентификаторами*. Они важны для согласованности и масштабируемости. Например, для Kafka необходимо обеспечить правильное разделение тем с использованием корпоративных идентификаторов, поскольку эти идентификаторы будут многократно применяться, когда потребители объединяют темы. Идентификаторы предприятия важны для API и архитектурного стиля REST, потому что они могут связывать ресурсы вместе с помощью гипермедиийных ссылок (<https://oreil.ly/on05->), позволяя потребителям переходить к соответствующему ресурсу.

Данные предметной области тоже могут иметь отношения с корпоративными данными и данными других областей. На уровне предприятия, например, таблица валют ISO может использоваться с кодами alpha-2, такими как US, NL, FR и т. д. В некоторых других областях используются коды alpha-3 со значениями, такими как USA, NLD, FRA и т. д. Хотя только одна из таблиц имеет классификацию предприятия, все же следует поддерживать связь между двумя таблицами в одном центральном и корпоративном решении MDM.

MDM и качество данных как услуга

Для успешного внедрения MDM на предприятии и в областях я рекомендую предлагать продукты MDM областям как услугу. Решения MDM часто бывают сложными, и их трудно реализовать. Абстрагирование от инфраструктуры и предоставление MDM как услуги может сильно упростить применение. Если вы используете централизованное решение, я рекомендую отделить области друг от друга. Если все основные и справочные таблицы хранятся централизованно, вы можете различать их с помощью метаданных и классифицировать, какие наборы данных являются данными предприятия и области.

То же самое относится к функциям управления качеством, например к профилированию, сопоставлению, стандартизации и проверке. Благодаря модели качества данных как услуги меры и контроль качества становятся прозрачными и областям не нужно реализовывать свои решения.

Курируемые данные

Управление основными данными частично совпадает с *курированием данных*: процессом сбора данных из различных источников и их интеграции, чтобы они стали более ценными, чем независимые части. В крупных организациях этот процесс обычно сочетается с DataOps, методологией создания и доставки тщательно отобранных, автоматизированных и надежных конвейеров данных. Курирование данных включает избавление от повторяющейся работы по интеграции для других команд. Это может быть важно для предприятий. Хотя курирование данных частично совпадает с MDM, важно показать различия и сформулировать строгие принципы для каждого из них. В следующих разделах я покажу различные подходы к созданию курируемых данных и управлению ими.

Цель MDM и создания курируемых данных — повышение ценности и облегчение жизни потребителей. Они используют методы ETL (извлечение, преобразование и загрузка), очистки данных, науки о данных и метаданных.

Как вы уже знаете, управление основными данными состоит из процессов, используемых для управления, централизации, организации, категоризации и владения данными. Во время этих процессов обычно удаляются дубликаты, данные исправляются и неверные данные удаляются. MDM уделяет большое внимание объединению сущностей (созданию одинаковых сущностей) и сокращению кластеров (созданию единой золотой записи). Результаты, полученные MDM, становятся авторитетными источниками основных и справочных данных. Уникальные данные обычно обнаруживаются централизованно, а выходные данные сохраняются в новом физическом местоположении, потому что они включают множество улучшений. Самое главное — новые данные не создаются в другом контексте. Факты, такие как количественная информация, остаются в собственном контексте, но для большей точности и согласованности производятся улучшения.

Курирование данных отличается от MDM, потому что не предполагает явно изменения существующих или создания новых данных. В этом отношении контекст тоже может быть изменен. Также курирование не предполагает явно сохранения данных. Результаты курирования могут быть представлены виртуально или как передовая практика, например, путем передачи метаданных, аннотаций или фрагментов кода через каталог данных. Поэтому курирование данных тесно связано с управлением метаданными. По большей части курирование связано с организацией метаданных, таких как информация о схемах, таблицах и столбцах, запросы, включая соединения и фильтры, и т. д.

Рассмотрим несколько подходов к обеспечению большей корпоративной согласованности архитектуры. Каждый имеет свои плюсы и минусы.

Обмен метаданными

Один из способов обеспечить большую семантическую согласованность данных — использовать общие метаданные. При таком подходе сами данные не изменяются и не передаются.

Организовать совместное использование метаданных разными областями можно несколькими способами. Метаданные можно доставлять вместе с данными, передавая их в дополнительном файле или внедряя в данные. Как вариант, метаданные можно хранить централизованно, например в каталоге данных. Метаданные должны содержать информацию о конкретных объектах данных, включая местоположения, аннотации, атрибуты, отношения и семантические значения. Если объекты равнозначны, они могут быть представлены аналогично с использованием меток или аннотаций. Это должно упростить потребление данных.

Процесс курирования данных с использованием метаданных можно ускорить. Курируя данные, эксперты по обработке данных обычно сотрудничают с экспертами в предметных областях. Другой подход — это краудсорсинг: приглашение сообществ пользователей и общественности для обработки данных. Оба подхода могут поддерживаться автоматическими инструментами аннотирования метаданными, которые сканируют данные и используют алгоритмы курирования для дублирования, классификации, оценки и прогнозирования принадлежности данных друг другу.

Интегрированные представления

Интегрированные представления — это предопределенные запросы, которые воссоздают данные с использованием тех же операторов SQL во время выполнения. Такой подход к обеспечению согласованности — общий для архитектур MDM и EDW. Интегрированные представления еще называют *виртуальными таблицами*. Потому что создается впечатление, что вы запрашиваете таблицу, но на самом деле это запрос, который сохраняется под покровом представления.

Разница в обеспечении семантической согласованности с помощью метаданных в том, что здесь представления могут содержать бизнес-логику. Они даже могут создавать или генерировать новые элементы данных. Эти изменения могут затрагивать не только синтаксические преобразования, но и преобразование контекста. Представления также могут быть *материализованными* (<https://oreil.ly/OHKUR>), то есть копирующими и сохраняющими результаты, что очень полезно для повышения производительности запросов.

Представления также можно комбинировать с метаданными. Они могут создаваться из метаданных и автоматически изменяться при изменении метаданных.

Повторно используемые компоненты и логика интеграции

Еще один способ сотрудничества и повторного использования данных — *совместное использование кода*. Здесь, помимо тщательно отобранных данных, передается основной код (фрагменты и сценарии) для генерации результатов и содействия эффективному повторному использованию. Этот код хранится в центральном и открытом репозитории, поддерживающем управление версиями, что позволяет командам DevOps вносить свой вклад и улучшать опубликованный код.

Такая модель хороша тем, что бизнес-логика применяется только внутри областей. Это позволяет командам отклоняться, вносить улучшения или использовать слегка оптимизированные версии логики по своему усмотрению. Кроме того,

эти выходные данные могут быть повторно сгенерированы по мере попадания улучшений сообщества в центральный репозиторий кода. Один из недостатков этой модели — согласованность. Разрешение командам изменять свой код может усложнить сравнение результатов между ними.

Повторная публикация данных

Некоторые поставщики БД выступают за создание курируемых данных путем сохранения и интеграции через ядра и платформы баз данных. При таком подходе команды DevOps являются одновременно и потребителями, и поставщиками данных. Они собирают как можно больше данных, извлекают и загружают их в хранилища, а также повторно публикуют или распространяют. Во время этого процесса семантический контекст может быть изменен, поэтому преобразования и улучшения должны применяться к существующим наборам данных. Еще можно создавать новые наборы данных. Этот подход частично совпадает с тем, как DDS распространяют интегрированные данные, что описано в главе 8.

Хотя эта модель доверия к определенным областям выглядит элегантно, в ней есть ряд рисков и проблем.

- Отслеживаемость и управление версиями, чтобы вы знали, что происходит с данными. Чтобы снизить риски прозрачности, попросите кураторов данных внести в каталог свои приобретения, а также последовательность действий и преобразований, применяемых к данным. Эти метаданные должны публиковаться централизованно.
- Качество данных и их управление. Фиксация данных в ядре создает риск того, что улучшения никогда не вернутся к исходным системам с золотым источником. Это приведет к отсутствию улучшений данных для транзакционных и операционных процессов. Есть вероятность возникновения проблем с правом собственности, потому что первоначальное право собственности на данные скрывается из-за нового владения данными.
- Операционные системы расширяются новыми функциями и компонентами приложений через ядро. Это означает, что их переходная целостность распространяется как на операционные системы, так и на ядро. Обеспечение согласованности транзакций становится труднее, ведь данные распространяются дальше, но также оптимизируются для соответствия нуждам потребителей и других вариантов использования. При неправильном проектировании могут возникнуть различия: появиться иерархические зависимости между ядром и операционной системой (-ми). Есть также риск появления служб, которые инкапсулируют данные как из переходной системы, так и из ядра. В результате может получиться сильно связанная архитектура.

Обеспечение семантической согласованности предприятия через повторную публикацию данных работает только тогда, когда есть четко определенные стандарты для моделей данных, происхождения и документации. Важно наличие и правил очистки данных, и процессов управления данными, определяющими, собираются ли данные и отправляются ли обратно на платформу.

Связь с управлением данными

Разница между курированием данных и управлением основными данными важна для управления данными. В MDM данные обычно корректируются по соображениям согласованности. MDM не должно изменять значение и создавать новые данные, поэтому владельцы данных и руководящие органы всегда могут проследить путь к владельцам данных исходных систем с золотыми источниками. С другой стороны, курирование данных может привести к смене владельца, потому что может включать создание или получение новых данных из существующих. Поэтому курирование данных в большей степени связано с областями и выполняется внутри них.

Различные подходы к созданию тщательно отобранных данных тесно связаны с демократизацией данных, что будет обсуждаться в главе 10. Масштабируемая архитектура переходит от групп центрального управления данными к децентрализованным группам. Увеличение прозрачности логики интеграции и возможность передачи данных обратно от одной команды другой, могут способствовать улучшению качества интерпретации данных и совершенствованию процесса принятия решений.

Итоги главы

Важность управления основными данными (MDM) очевидна. Пользователи могут принимать правильные решения, только если данные, которые они используют, согласованы и верны. MDM обеспечивает согласованность и качество на уровне предприятия — два аспекта, которые настолько тесно взаимосвязаны, что многие решения захватывают их оба.

Практический способ внедрить MDM в организации — начать с самого простого стиля реализации: репозитория. Используя его, вы без корректировки каких-либо операционных систем сможете быстро получить ценность, узнав, какие данные необходимо согласовать, или выявить данные плохого качества.

Следующий шаг — определение области видимости. Постарайтесь не попасть в ловушку унификации корпоративных данных, выбрав *все* данные. Начните

с предметной области, представляющей наибольшую ценность для организации. Например, с клиентов, контрактов, организационных единиц или продуктов. Отберите только самые важные поля. Количество атрибутов должно исчисляться десятками, а не сотнями. После того как вы пришли к соглашению, приведите в соответствие процессы и управление. Сделайте договоренности о сроках и пересмотрах понятными для всех. Не забывайте о работе с метаданными, чтобы основные данные были каталогизированы, а пользователи знали, какие элементы данных из каких систем-источников подходят им и как эти элементы проходят через конвейеры данных.

Последний шаг и конечная цель — достичь сосуществования: когда улучшения перетекают прямо в системы-источники. Этот шаг самый сложный, потому что требует внесения множества изменений в архитектуру. Системы-источники должны быть способны обрабатывать исправления и улучшения основных данных из хранилища основных данных. Эти изменения должны распределяться соответствующим образом с использованием шаблонов архитектуры интеграции.

Мы также обсудили повышение согласованности данных на предприятии за счет создания курированных данных. Для этого важны эффективная модель коммуникации и управление данными. Организация становится более эффективной, когда поощряется повторное использование данных. Вот почему многие компании обращаются к подходам, основанным на взаимодействии с сообществом, к моделям открытых данных и рынкам данных. Мы обсудим это в главе 10.

ГЛАВА 10

Демократизация данных с помощью метаданных

В этой главе мы подробнее разберем роль метаданных в архитектуре и способы управления ими. *Метаданные*, как мы уже знаем, описывают все соответствующие аспекты новой архитектуры. Они связывают все воедино и являются ключом к обеспечению понимания, контроля и эффективности, к которым стремятся крупные предприятия.

Метаданные еще и сложный объект для управления, ведь они разбросаны по множеству инструментов, приложений, платформ и сред. Как правило, в большой архитектуре данных существует множество организованных репозиториев метаданных. Большинство метаданных также тесно связаны с конкретным продуктом поставщика. Их объем и разнообразие огромны. Поэтому, прежде чем ими можно будет управлять, обычно нужно правильно выбрать, организовать и внедрить метаданные. В этой главе вы узнаете, на чем следует сосредоточиться, и познакомитесь с основными составляющими хорошей корпоративной модели метаданных.

Вам нужно будет автоматизировать сбор, обнаружение, обслуживание и использование метаданных, чтобы сделать их неотъемлемой частью архитектуры. Мы рассмотрим шаблоны интеграции и обсудим, почему определенные метаданные должны храниться централизованно (чтобы не изобретать колесо заново, лучше повторно использовать как можно больше шаблонов интеграции).

И последнее, но не менее важное: мы рассмотрим демократизацию данных, графы знаний и способы использования метаданных для управления данными. Чтобы облегчить этот процесс, метаданные должны быть повсеместными на уровне предприятия, открытыми и доступными для всех. Это единственная центральная модель данных, которую мы будем использовать.

Управление метаданными

Термин «*метаданные*», определяемый как «данные о данных», был придуман в 1990-х годах. Тогда многие поставщики активно подключились к метаданным, предоставив инструменты для более эффективного управления данными и моделями БД. Большинство таких инструментов также предоставляют возможности прямого инжиниринга для автоматического создания моделей данных приложений. В этом подходе концептуальные и прикладные логические модели данных, которые описывают концепции и зависимости, используются для автоматического создания физических моделей данных приложения. Сюда входят таблицы, столбцы, отношения внешних ключей и типы данных. Еще эти инструменты позволяют позаботиться об операционных аспектах и определить, какие данные должны быть активными, а какие должны быть заархивированы.

В эпоху больших данных и альтернативных технологий, таких как NoSQL и продвинутая аналитика, управление метаданными стало более динамичным и разнообразным, с различными формами и очертаниями метаданных. Скорость изменений и разнообразие решений тоже полностью отличаются от традиционных архитектур. Сегодня метаданные можно найти везде: в приложениях, БД, технологиях интеграции данных, управлении основными данными, облачной инфраструктуре и т. д. Поэтому они более разрознены: у каждой платформы и инструмента есть собственный каталог метаданных и репозитории. Рисунок 10.1 иллюстрирует эту проблему.



Рис. 10.1. Метаданные часто очень разбросаны. Как правило, на большинстве предприятий существует множество организованных репозиториев метаданных

Такая разрозненность усложняет использование метаданных. В разрозненной и распределенной среде сложно контролировать, местоположение, перемещение и значение данных. Чтобы решить эту проблему, нужно связать метаданные по разным измерениям: создать представление метаданных в масштабе всего предприятия.

Мой опыт показывает, что процесс интеграции, автоматизации и объединения метаданных на предприятии требует стандартизации, продуманного выбора и процедур. Распространение технологий с несовместимыми возможностями метаданных препятствует стремлению к созданию контролируемой среды. Поэтому необходимо сделать тактический выбор, чтобы объединить несколько решений, применить автоматизацию и определить, какие метаданные наиболее актуальны.

Я рекомендую сосредоточиться на трех основных целях.

1. Построение корпоративной модели метаданных, которая представляет все области предприятия, их атрибуты и взаимосвязи. Она может использовать концептуальную модель и связанные данные для предоставления нескольких методов динамического взаимодействия.
2. Определение критически важных коллекций метаданных и наилучшего архитектурного подхода для сбора или предоставления метаданных через API, оболочки и потоки.
3. Демократизация (мета-) данных через возможности самообслуживания и порталы.

В следующих разделах я рассмотрю каждую из этих целей более подробно.

Модель метаданных предприятия

Хорошая стратегия управления метаданными развивается органически. Все начинается с малого и простого. Она поддерживается верным планом, четкими процессами и правильным выбором архитектурных блоков. Ваш план — это интегрированная и унифицированная модель, которая работает во всех дисциплинах и технологиях управления данными. Главное здесь не сосредоточиться на узком каталоге или решении для коммерческих метаданных, а взглянуть на более широкую картину и определить основные сущности в рамках корпоративной модели метаданных.

Я не прошу создать модель корпоративной БД, но подумайте о том, что объединяет вашу организацию. Спросите себя: какие бизнес-метаданные важны? Какие технические метаданные требуются для взаимодействия? Какие процессы и потоки собирают данные? Где создаются и поддерживаются модели или схемы? Какая информация должна предоставляться централизованно, чтобы отдел управления данными мог правильно выполнять свою работу? После анализа этих вопросов нужно наметить жизненный цикл каждого из потоков метаданных и определить все зависимости. В итоге вы получите не зависящую от поставщика унифицированную модель метаданных, которая может соединить организационные единицы, процессы, технологии и данные.

DATAOPS ИМЕЕТ ТЕСНУЮ СВЯЗЬ С МЕТАДАННЫМИ

DataOps — передовая практика совместного управления данными, направленная на повышение эффективности связи, интеграции и автоматизации потоков данных между командами в организации. Речь также идет о том, чтобы все «говорили на одном языке» и договаривались о том, какие данные есть, а каких — нет.

Метаданные обеспечивают эту семантику и широкое понимание того, что означают данные. Таким образом, если вы хотите освоить DataOps, убедитесь, что вы также знаете, как управлять метаданными.

Я рекомендую не вносить слишком много изменений и тем самым быстро увеличивать сложность. Начните с основ: списки приложений, источники надежных данных (золотые источники), схемы БД и интерфейсов, владение данными и безопасность. Постепенно расширяя область видимости, вы сможете контролировать управление метаданными. Важно работать над согласованностью и обеспечивать правильную интеграцию одних метаданных с другими. Для организации метаданных можете использовать реестр, управляемый онтологией (см. врезку «Что такое тезаурус и в чем разница между таксономией и онтологией» на с. 322).

Эталонная модель, показанная на рис. 10.2, не является обязательным или исчерпывающим списком. Это просто рекомендуемая инфраструктура, показывающая, на каких метаданных сосредоточиться. Ее можно использовать как отправную точку, приспосабливая под нужды и потребности своей организации. В следующих разделах мы более подробно рассмотрим различные части этой инфраструктуры.

Наборы и объекты данных

В основе фреймворка лежат наборы данных. В главе 7 вы увидели логическую модель управления ими. Наборы данных абстрактны. Это самый важный тип значений, которые нужно регулировать. Они могут быть связаны с (физическими) объектами данных, процессами, приложениями, инфраструктурой, владельцами, бизнес-терминами и т. д.

Объекты и атрибуты данных представляют собой наборы данных и часто хранятся физически. На верхнем уровне находятся объекты данных. Они могут быть связаны с базами данных, файлами, API или событиями. Уровнем ниже располагаются атрибуты. Это свойства, связанные с объектами данных. Они тоже могут быть связаны с элементами данных. Мы обсуждали это в главах 6 и 7. В них вы узнали, что элемент данных, например `household`, может быть связан с различными физическими атрибутами данных.

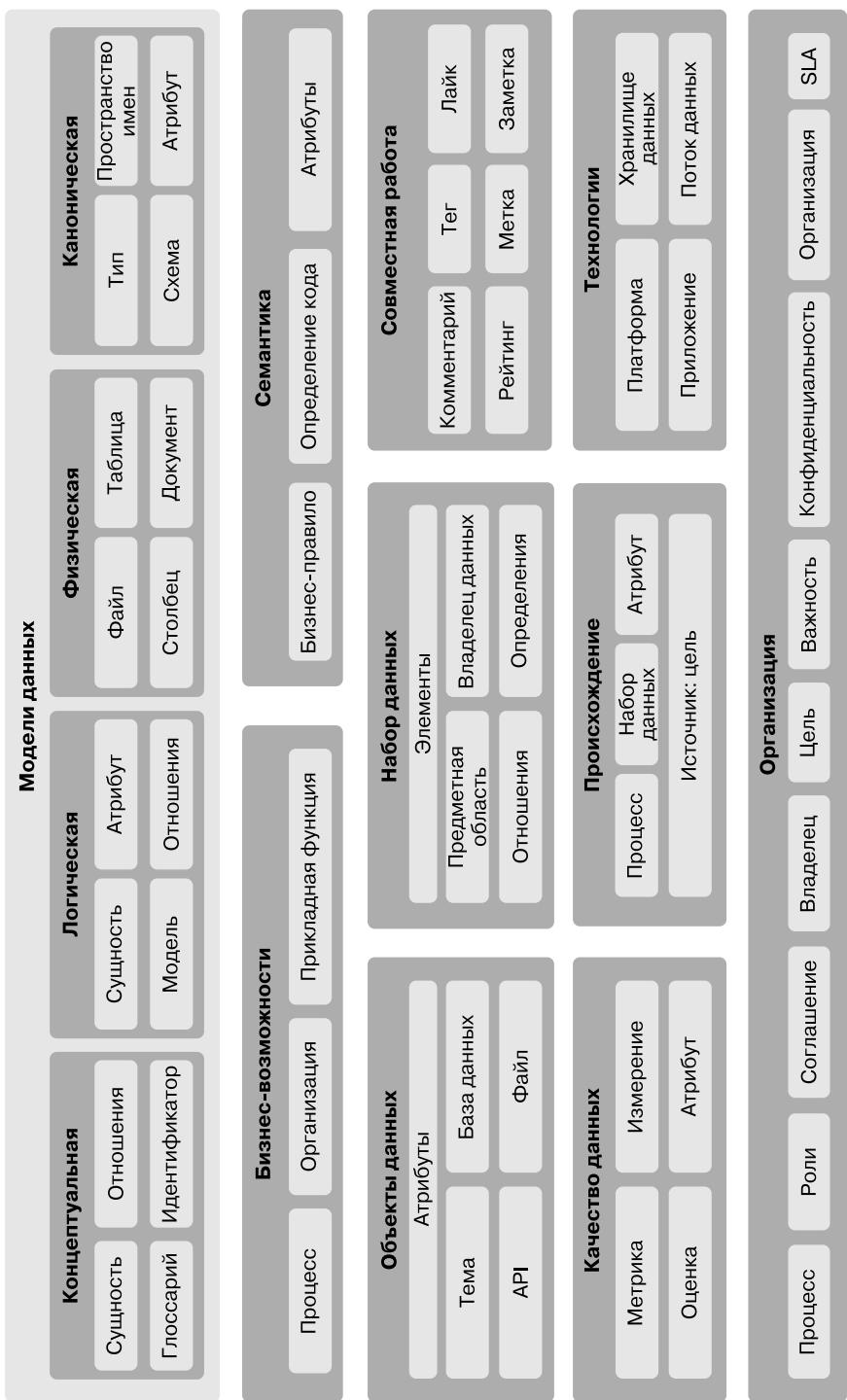


Рис. 10.2. Основные области управления метаданными для предприятия. Этот список областей не является исчерпывающим. Каждая организация использует их по-разному

Модели данных

Уделим особое внимание моделям данных, находящимся в верхней части инфраструктуры.

На самом высоком уровне находятся *концептуальные модели данных*, которые могут включать онтологии и таксономии. Они обобщают важнейшие бизнес-концепции и передают значение и цель данных с точки зрения бизнеса. Их можно оставить абстрактными, но я рекомендую снабдить все термины определениями, атрибутами и зависимостями. Определения должны учитывать ограниченные контексты поставщиков и потребителей данных и помогать организации лучше понять, что означают данные в конкретном контексте.

ПОЧЕМУ КОНЦЕПТУАЛЬНЫЕ МОДЕЛИ ДАННЫХ ЧАСТО ОТСУТСТВУЮТ

Есть несколько причин отсутствия концептуальных моделей данных. Одна из них в том, что многие бизнес-пользователи и IT-специалисты не знают о них. Многие концептуальные модели создаются неявно, например в виде эскиза на доске или в документе, но не хранятся в цифровом виде. Концептуальные модели часто путают с другими моделями данных. Редко встречаются люди с опытом и глубокими знаниями в области концептуального моделирования. Другие считают концептуальное моделирование данных слишком абстрактным или сложным. Захват моделей затруднен и из-за отсутствия хороших, простых в использовании инструментов для захвата всех концептуальных моделей данных в корреляции со всеми приложениями. Большинство инструментов охватывают лишь узкий круг технологий. Protégé (<https://protege.stanford.edu/>) — популярный инструмент с открытым исходным кодом для построения концептуальных моделей данных с использованием OWL. Его главный недостаток заключается в том, что его довольно сложно использовать без соответствующей подготовки.

Еще одна причина частого отсутствия концептуальных моделей данных — люди сомневаются в добавочной ценности создания этих моделей. В рамках процессов гибкой разработки или в условиях нехватки времени я видел, как люди решали вообще не фиксировать какие-либо неявные мысли или концепции. В таких ситуациях бизнес-пользователи, разработчики и инженеры идут короткими путями для своевременной доставки приложения.

Уровнем ниже находятся *логические модели данных приложений*, которые развиваются из концептуальных моделей. Основные концепции оставляют место для сущностей и атрибутов. Если все сделано правильно, они должны объяснить происхождение бизнеса и то, как концепции были переведены в логический дизайн БД приложения. Обычно метаданные этого типа можно получить с помощью инструментов моделирования данных.



Логическое проектирование приложений и БД – первый конкретный шаг к выбранной вами технологии. Это означает, что выбор технологии существенно влияет на проектирование. При выборе реляционной БД логическая модель будет содержать таблицы, столбцы и ключи базы данных. Для документов XML модель будет содержать элементы, атрибуты и ссылки XML.

На самом нижнем уровне находятся *физические модели данных приложения*: схемы, точно определяющие, какие данные в каких базах хранятся. Этот тип метаданных обычно автоматически извлекается из инструментов моделирования данных или баз данных, работающих на физических платформах.



Требования к безопасности, целостности и производительности очень важны в разработке схем базы данных. На логическом уровне может существовать связь между двумя объектами, но по соображениям производительности эти два объекта могут храниться в одной таблице. Поэтому физическая структура БД может сильно отличаться от логической модели. Это еще означает, что вы можете начать с одинаковой логической модели данных приложения, но прийти к совершенно разным физическим реализациям. Например, вы можете использовать одну и ту же логическую модель для реализации транзакционной системы или системы отчетности, но каждая будет использовать свой собственный физический проект.

Без всех этих типов метаданных ориентированные на данные проекты не смогут найти нужные данные. Их отсутствие значительно усложнит интеграцию данных, ведь не будет присутствовать информация о значении данных и об их создании в исходной системе.

Дilemma концептуальных, логических и физических моделей и их взаимосвязей в том, что еще не все инструменты охватывают все пространство типов БД. DBMS Tools (<https://oreil.ly/Bzjd5>) поддерживает подробный список всех моделей БД и баз, с которыми они работают. Ни один инструмент моделирования баз данных не поддерживает все БД. Автоматическая прямая разработка подходит только для некоторых типов баз данных, а для концептуальных моделей данных обычно нужны другие инструменты. Я рекомендую стандартизировать некоторые из ваших инструментов моделирования данных и типы баз, но хранить высоконивневые или абстрактные версии моделей данных в центральном репозитории метаданных. Это дает командам гибкость в выборе инструмента моделирования данных, а группе управления – возможность получать аналитические данные.

Другой подход к сбору и накоплению метаданных, описывающих данные, заключается в использовании того, что я называю *агентами*. Это автоматизированные

инструменты или механизмы обнаружения, которые автоматически сканируют, рассматривают и собирают метаданные. Такой подход еще известен как обнаружение метаданных (<https://oreil.ly/bYi4s>). Он дает возможность преобразовать физические модели в логические и потом в концептуальные. Такое связывание позволяет быстрее находить данные. Расширяя этот подход с помощью аналитики, такой как машинное обучение, вы можете лучше предугадывать, какие пользователи хотели бы видеть или открывать значения скрытых данных.

Последний подход к сбору и отсутствию метаданных — *краудсорсинг*, основанный на использовании знаний сообщества для масштабирования процесса сбора недостающей информации с использованием простых и интуитивно понятных инструментов. Мы еще вернемся к этому аспекту в подразделе «Управление данными и безопасность» на с. 319.

Подходы на основе экспорта, сбора и краудсорсинга метаданных можно идеально сочетать и смешивать. Например, подход «снизу вверх» к сбору и созданию исходных концептуальных моделей может быть дополнен предоставлением владельцам данных и приложений возможности вручную вносить улучшения для получения конечных результатов.

Преимущество регистрации метаданных приложений и БД в том, что бизнес-термины и их переводы в базовые структуры баз данных становятся понятными для организации. Это должно позволить техническим группам быстро оценивать новые требования, реагировать на них и решать проблемы, связанные с данными. К тому же это позволяет им повторно использовать существующие знания и помогает в проектах миграции и интеграции данных.

Связывание всех метаданных из моделирования и проектирования данных обеспечивает интеллектуальную интеграцию и использование. Если бизнес-метаданные связаны с техническими, вы знаете, где хранятся и расположены физические объекты данных и их атрибуты. Теперь вы можете автоматически извлекать эти данные, объединять и интегрировать их и передавать потребителям. Этот подход еще известен как интеграция данных на основе онтологий (https://oreil.ly/9_YyJ).

Модели интерфейсов

Следующий логический шаг — фиксация всех моделей интерфейсов, включая проекты интерфейсов, номера версий, информацию о происхождении, шаблоны доставки, расписания и т. д. Они имеют прочные отношения с контрактами на поставку и совместное использование данных.

Сбор метаданных, их хранение в центральном репозитории интерфейсов и экспортование помогут всем вашим разработчикам. Без этих метаданных разра-

ботчики не будут знать, как функционируют интерфейсы, какие данные доступны и какие интересные тенденции лежат в основе каждого интерфейса. Это позволяет им тестировать и проверять совместимость, что гарантирует безопасную подписку потребителей данных. Наличие метаданных интерфейса повышает осведомленность и увеличивает повторное использование.

Информация о происхождении и ее интеграция

Информация о происхождении описывает происхождение данных и их перемещение с течением времени, как мы узнали из подраздела «Происхождение и перемещение данных» на с. 217. Информация о происхождении играет важную роль, показывая зависимости приложения и движение данных по цепочке поставок. Благодаря ей мы можем повторно использовать логику преобразования. Используя обнаруженные отношения, статистику и вариации, можно использовать информацию о происхождении, чтобы определить влияние качества данных в системе-источнике на разных потребителей данных.

Чтобы зафиксировать информацию о происхождении, можно использовать те же подходы к экспорту, что и для моделей данных. Любой из них уже поддерживается коммерческими продуктами, обычно за счет использования базовых репозиториев метаданных. Информацию о происхождении можно собрать и вручную, выгрузив или передав ее. Команде, использующей редкий инструмент ETL или язык программирования, может потребоваться приложить дополнительные усилия, чтобы передать информацию о происхождении в централизованное хранилище.

Управление данными и безопасность

Метаданные описывают данные и их движение, но они еще должны определять, как данные используются и управляются. В главе 7 вы узнали о владении данными, соглашениях об их совместном использовании, пользователях, ролях, классификациях, метках целей и правилах управления, связанных с метаданными. Вы увидели логический план правильного управления всем вышеперечисленным. Эти метаданные — основа для автоматического создания и применения политик безопасности, что помогает защитить данные. Рассмотрим самые важные типы метаданных.

- *Метаданные с информацией о принадлежности и золотых источниках.* В главе 7 мы обсудили метаданные, которые нужны для представления всех уникальных наборов данных, включая информацию об их принадлежности, в рамках всей организации. Это хранилище называется *списком золотых источников*.

- *Совместные метаданные*, получаемые в сотрудничестве, могут добавить большую ценность. Сделав данные видимыми, поделившись знаниями и разрешив пользователям сотрудничать, вы можете создать сообщество. Примеры совместных метаданных включают формы обратной связи, рейтинги, закладки и комментарии.
- *Метаданные, описывающие качество данных*, являются предпосылкой для многих ключевых бизнес-процессов и аналитических моделей. Они определяются по разным измерениям качества данных, таким как точность, актуальность и полнота. Эти метрики и оценки обычно связаны с характеристиками отслеживаемых атрибутов физических данных.
- *Метаданные бизнес-архитектуры*. Чтобы задать контекст данных, важно связать метаданные с архитектурой вашего бизнеса. Вы можете сделать это с помощью бизнес-возможностей, организационных структур, таких как подразделения и отделы, процессов, приложений и общих функций приложений. Эта информация позволит вам сопоставить данные с моделями проектирования, ориентированными на предметную область, и определить, какие данные лежат в основе каких процессов.
- *Метаданные технологий* включают в себя все метаданные, которые нужно связать с конкретной технологией: серверные платформы, приложения, потоки данных, хранилища данных, инфраструктуру и т. д. Связь с техническими платформами и возможностями помогает ИТ-отделу видеть влияние обслуживания, управления версиями и поставщиками, а также управления проблемами на управление данными по мере того, как компания выполняет обновления и изменения системы.
- *Семантика метаданных*. Бизнес-правила, касающиеся отношений и зависимостей, сложны. Они обычно управляются в центральных приложениях с использованием четко определенных терминов в качестве архитектурных блоков для описания зависимостей между данными, процессами и приложениями. Управление ими как метаданными может помочь вам увидеть, какая контекстная информация содержится в этих правилах и как она соотносится с другими областями.
- *Аналитические метаданные*. Последняя область — это метаданные, которые поставляются с инструментами бизнес-анализа и аналитики: метаданные отчетов, управления моделями и т. д. Эти метаданные также могут включать информацию о происхождении, показывающую перемещение данных через серию заданий или небольших преобразований из источников в отчеты и аналитические функции. Сбор этой информации дает представление о том, какие детальные данные использовались для составления отчетов об измерениях и цифрах.

Метаданные и управление ими могут принести большую пользу вашей организации. Чем эффективнее вы справитесь с этим, тем больше информации вы получите. Это может снизить затраты за счет ускорения разработки и сокращения периодов обслуживания. Это помогает обеспечить соответствие и безопасность всей архитектуры.

Организация метаданных — сложная задача, но не все метаданные одинаковые. Одна из важных характеристик — ссылочный характер. Метаданные часто однозначно идентифицируют объекты и ресурсы, которые могут быть связаны и включают некоторое семантическое описание. Такая информация может стать основой для расширенных сервисов, включая поиск, обслуживание и автоматическую доставку. Вот почему в следующих разделах онтологии и семантическая сеть рассматриваются как часть идеи масштабного управления метаданными.

Граф корпоративных знаний

Способ связывания объектов и ресурсов похож на работу Всемирной паутины: гипертекстовые ссылки могут связывать что угодно с чем угодно. Гиперссылка может относиться к другой странице, определенному разделу веб-страницы, изображению или даже файлу. Поисковые системы, наблюдающие за гиперссылками, могут легко перемещаться по всему этому.

В 2000 году Тим Бернерс-Ли (<https://oreil.ly/bY511>), один из изобретателей Всемирной паутины, опубликовал статью, в которой говорится, что все может быть связано. Расширяя текущий стандарт интернета с помощью структуры семантической сети (<https://oreil.ly/d0LYh>), любые данные можно совместно и повторно использовать в рамках любого приложения, предприятия или сообщества¹. Разнообразные технологии позволяют кодировать семантику с данными. Давайте кратко рассмотрим некоторые из них.

- *Структура описания ресурсов (resource description framework, RDF)*. Это технологическая структура для представления информации о ресурсах в форме графа. Она позволяет распределить и децентрализовать данные, но ее можно использовать для запроса и объединения данных с помощью протокола HTTP. Данные RDF — это бессхемный дизайн с коллекциями триплетов, которые можно легко расширить и представить.

¹ Термин «семантическая сеть» относится к видению W3C сети связанных данных. Технологии семантической сети (<https://www.w3.org/2001/sw>) позволяют людям формировать хранилища данных в сети, конструировать словари и писать правила для обработки данных.

- **Язык веб-онтологий (*Web ontology language, OWL*)**. Это спецификация, которая добавляет онтологические возможности к RDF. Она точно определяет, что вам нужно написать с помощью RDF, чтобы получить действительную онтологию. OWL также предоставляет полезные выражения и аннотации для внедрения моделей данных в реальный мир. OWL считается языком концептуального моделирования, но из-за того, что эта спецификация связывает и соотносит наборы данных, она еще является и логической моделью данных.
- **Протокол SPARQL и язык запросов RDF (SPARQL)** (<https://oreil.ly/5eHVY>). Это язык запросов, используемый для извлечения и обработки данных, хранящихся в RDF. Он может собирать запросы из объединенных источников данных и включать мультимодельные графовые БД для поддержки нескольких моделей данных в единой интегрированной серверной части.
- **Язык описания контурных ограничений (*Shapes Constraint Language, SHACL*)**. SHACL — это язык для описания и проверки графов RDF. С его помощью вы можете взять граф контуров и граф данных и сравнить их друг с другом.
- **Простая система организации знаний (*Simple Knowledge Organization System, SKOS*)**. Это общая модель данных для обмена и связывания систем организации знаний через интернет. Ее можно использовать для определения систем организации знаний, таких как тезаурусы¹, схемы классификации, системы предметных заголовков и таксономии.

ЧТО ТАКОЕ ТЕЗАУРУС И В ЧЕМ РАЗНИЦА МЕЖДУ ТАКСОНОМИЕЙ И ОНТОЛОГИЕЙ

Таксономии — это самый простой вариант. Они содержат только термины, организованные в иерархическую структуру. Обычно это простые комбинации классов, которые не определяют типы отношений между сущностями или уровень их иерархии в таксономии. Например, в биологии таксономия может использоваться для классификации животных как травоядных, всеядных, плотоядных и т. д.

Тезаурусы добавляют к каждому понятию неиерархические отношения между концепциями и другими свойствами.

Онтологии находятся на тяжелом конце спектра. Они больше касаются отношений между объектами. *Онтологии* — это иерархическая структура, которая может иметь внутренние ограничения на отношения и разделять сущности на классы, каждый со своими собственными установленными правилами. Например, система классификации для документирования всех компонентов и характеристик самолета.

И таксономии, и онтологии могут храниться и визуализироваться в виде *графа знаний* или набора объектов, где где типы и свойства имеют объявленные для них значения и связаны отношениями, образуя узлы и ребра между ними, что и делает граф знаний графом.

¹ Тезаурусы — это книги, в которых слова перечислены в группах синонимов и связанных понятий.

Стандарты семантической сети — это новые стандарты де-факто для взаимодействия на основе онтологий и публикации контента в Сети. Крупные технологические компании, такие как Facebook, Google, Microsoft и Amazon, приняли и применили их принципы и методологию. Прелесть этого стандарта в том, что тот же набор стандартов семантической сети может применяться в управлении данными. Комбинируя возможности управления на основе онтологий, можно создать открытый граф корпоративных знаний¹. *Граф корпоративных знаний* — это БД, которая представляет и объединяет все знания и взаимосвязи организации (предметной области). Это набор ссылок на ресурсы, контент, метаданные и данные вашей организации, которые используют модель данных для описания людей, мест и сущностей и их взаимосвязи. Этот репозиторий может содержать как можно больше релевантных метаданных и может быть дополнен контекстной и семантической информацией.

Одна из замечательных особенностей — возможность напрямую связать граф знаний с данными. Таким образом, граф знаний используется для соединения и представления информации о данных с использованием графоподобной структуры. Это позволит автоматически заполнять и создавать графы или находить соответствующие данные на основе определенной онтологии. Запросы и объединение распределенных данных работают, пока вы придерживаетесь отраслевых стандартов. Используя этот подход, вы можете увидеть в одном месте всю связанную с данными информацию, такую как семантика, возникновение, происхождение, фазы жизненного цикла, владение и соответствующие правила, готовую для глубокого анализа. Такие компании, как Cambridge Semantics (<https://oreil.ly/p6Pqg>), также работают над этой идеей, но все еще существует много пробелов, которые необходимо заполнить.

Еще один интересный аспект этой идеи — возможность настроить графы знаний так, чтобы они частично поддерживались самими областями. Другая часть по-прежнему будет обслуживаться централизованно. Нужно установить стандарты уровня предприятия для наиболее важных объектов метаданных и их взаимосвязей. Эта модель предприятия составляет основу вашей модели.

По периметру вы можете позволить другим областям расширять центральную модель метаданных без ущерба для базовой модели (рис. 10.3). Например, область маркетинга может добавить в модель свои собственные предметно-ориентированные сущности для детализации модели метаданных, чтобы удовлетворить их потребности. Они могут даже развернуть собственное хранилище

¹ К сожалению, инструментов для поддержки и распространения онтологий не так много. Anzo (<https://oreil.ly/QvFL1>), Enterprise Architect (<https://sparxsystems.com/>) и Poolparty (<https://www.poolparty.biz/>) — одни из самых популярных коммерческих вариантов. Кроме того, вы можете просмотреть список хостов W3C (<https://oreil.ly/wC3bd>).

метаданных и сделать его общедоступным. Все метаданные легко могут расширяться, объединяться и запрашиваться вместе, если репозитории имеют надлежащим образом стандартизованные (OWL) конечные точки и встроенные связи с местоположениями ресурсов (данных).

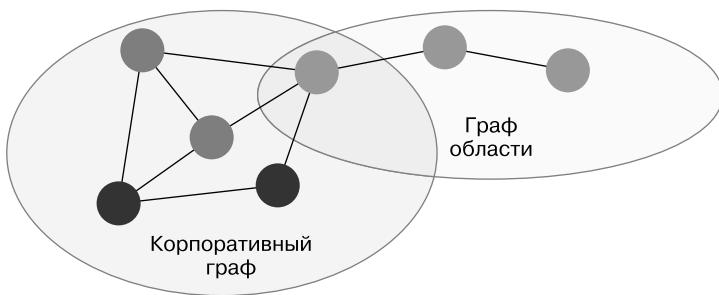


Рис. 10.3. Граф корпоративных данных — это сердце общей модели метаданных. Другие области могут расширять модель своими локальными сущностями, чтобы упростить решение своих задач

Мы можем расширить идею использования стандартов семантической сети для управления метаданными, чтобы полностью связать их с нашими корпоративными моделями. При таком подходе все что угодно, от управления данными, можно связать с чем угодно. Например, центральные классификации конфиденциальности данных можно связать с концептуальными терминами, и, поскольку у вас есть базовая ссылка на конечные точки данных приложения, вы знаете, какие атрибуты данных являются конфиденциальными. Граф знаний в качестве большой семантической сети сущностей и их атрибутов позволяет находить самые подходящие сущности, основываясь на семантическом сходстве между ними. Вы можете разрешить областям расширять свои предметно-ориентированные графы знаний с помощью предметных словарей, таксономий или онтологий, как показано на рис. 10.3.

Хотя подход, основанный на онтологиях, сложно воплотить, а инструментов мало, он может сильно улучшить общую согласованность ландшафта данных. Поэтому важно подготовиться к такому новому развитию. Его преимущество в том, что он использует те же открытые стандарты, что и интернет. Внутренние знания и значения, связанные с данными, становятся доступными каждому. Крупные технологические компании часто интегрируют эту технологию с искусственным интеллектом, машинным обучением и обработкой естественного языка. Вы даже можете связать эту идею с дисциплинами интеграции или безопасности данных. Например, можно интегрировать и преобразовывать данные автоматически на основе семантического контекста и предоставленных отноше-

ний или динамически изменять доступ к данным на основе описаний процессов, задокументированных в одном из графов знаний. Помните интеллектуальную службу потребления, о которой мы говорили в главе 3? Выполнение упрощенных синтаксических преобразований данных с использованием метаданных может помочь значительно упростить интеграцию и использование данных.



Для интеграции и использования данных с помощью интеллектуальной службы потребления нужен семантический преобразователь (<https://oreil.ly/CqJMc>). Это приложение, которое помогает преобразовывать элементы данных из одного атомарного объекта данных в другой. Средство семантического отображения в этом подходе находит соответствующие элементы данных (метаданные) через семантические отношения, которые становятся явными с помощью онтологий. Интеллектуальная служба потребления использует эти метаданные в качестве входных и преобразует их в запрос, который относится к базовым физическим источникам данных. На выходе получаются настоящие физические данные, которые могут быть доставлены потребителям несколькими способами: файлы, представления базы данных, темы, API и т. д.

Я рекомендую использовать подход, основанный на графах знаний, — начинать с малого. Не пытайтесь создавать огромные диаграммы со всеми связями, которые поймет только небольшая группа технических специалистов. Создавайте модель медленно и сделайте ее доступной и для бизнес-пользователей, чтобы каждый извлек выгоду из нового понимания. В конечном состоянии вы можете применять расширенные алгоритмы и дополнительные инструменты к графикам знаний, чтобы обеспечить интеллектуальное потребление данных, улучшить общее качество и обнаружить скрытые закономерности.

Архитектурные подходы к управлению метаданными

Одна из самых больших проблем — единообразное представление большого количества метаданных. Поставщики используют неидентичные схемы БД и собственные структуры данных для хранения метаданных и презентаций в различных формах. У них разные целевые области. Здесь нам нужен *уровень метаданных*, который может использовать и унифицировать метаданные для обмена между сторонами. Прежде чем мы определим, как решить эту проблему, давайте сначала рассмотрим различные архитектурные шаблоны.

- *Консолидированные метаданные.* При *консолидированном* (централизованном) подходе все метаданные объединяются в единый централизованный

репозиторий или базу данных. Метаданные в этой модели постоянно синхронизируются, или события доставляются в центральную систему по мере внесения изменений. Такой подход более тесно взаимосвязан и нужен только тогда, когда определенные метаданные должны управляться централизованно. Безопасность, обзоры приложений, регистрация прав собственности на данные и контракты обычно требуют централизованного внимания и лучше работают с централизованно управляемыми хранилищами метаданных. Эта модель важна и по нефункциональным причинам: например, извлечение происхождения из множества разных хранилищ может вызвать огромную нагрузку на сеть. Наличие центральной копии могло бы решить эту проблему.

- *Федеративные метаданные.* При использовании *федеративного*, или децентрализованного, подхода к метаданным все метаданные хранятся и управляются локально и собираются вместе только при необходимости. Для создания в реальном времени представлений всех федеративных хранилищ метаданных можно использовать REST API и конечные точки RDF. Обратная сторона такого подхода — если одно из хранилищ федеративных метаданных не работает, то централизованные представления могут оказаться неполными. Кроме того, должны быть унифицированы по-разному представленные конечные точки, что требует дополнительного уровня абстракции. Я объясняю это в следующем разделе.
- *Общие метаданные.* *Общий*, или гибридный, подход сочетает преимущества двух предыдущих подходов. Важные элементы метаданных собираются и объединяются централизованно, а другие, менее важные метаданные управляются локально и децентрализованно разными командами. Преимущество этого подхода в возможности значительно уменьшить издержки на отображение и поиск метаданных, а наиболее важные метаданные по-прежнему можно контролировать централизованно.

Эти подходы можно сочетать и смешивать в рамках гибридной архитектуры метаданных. Например, важные метаданные могут храниться и контролироваться централизованно с использованием консолидированного подхода, а менее важные метаданные — управляться децентрализованно областями с использованием федеративного подхода.

Совместимость метаданных

Среди решений конкурирующих поставщиков, ориентированных на управление данными, почти нет стандартов для взаимодействия метаданных. Большинство репозиториев имеют собственные закрытые API. Некоторые из них могут предо-

ставлять определенные функции для синхронизации или расширения модели метаданных, но модель по-прежнему либо закрыта, либо исправлена. Настройка в более крупном масштабе очень сложна.

ИНТЕГРАЦИЯ МЕТАДАННЫХ

Интеграция метаданных похожа на интеграцию любых других данных. Для физического извлечения, связывания в реальном времени или событийно-ориентированной интеграции используются те же шаблоны взаимодействия и интеграции. Некоторые репозитории могут одновременно реализовывать несколько шаблонов интеграции. Для связывания в реальном времени обычно используются API, для распределения метаданных между локальными и центральными репозиториями можно использовать потоковую передачу. При более интенсивном обслуживании данных следует использовать шаблоны RDS, например, для анализа сложных графов, тяжеловесного агрегирования или множественных соединений. Неудивительно, что оптимизированная графовая база данных лучше подходит для хранения информации о происхождении, в то время как для частого поиска владельцев данных может быть достаточно хранилища данных типа «ключ — значение». Кроме того, вы можете расширить архитектуру метаданных с помощью механизмов поиска и индексирования, чтобы упростить и повысить эффективность сложных запросов.

Из-за всех этих вариаций, их разных масштабов и отсутствия стандартов взаимодействия вам придется интегрировать множество метаданных. Реализация подхода федеративных метаданных, например, означает, что вам необходимо создать слой оболочки (рис. 10.4), чтобы скрыть все различия API и обеспечить единообразный доступ ко всем базовым источникам метаданных.

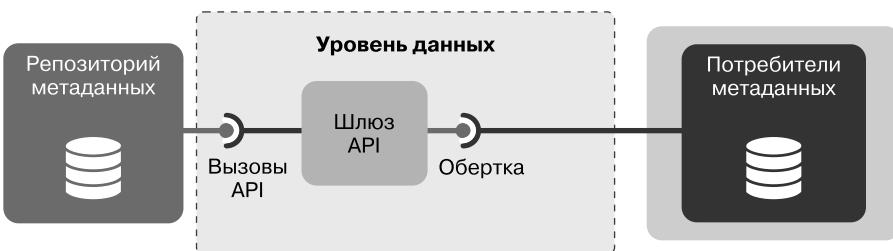


Рис. 10.4. Обмен метаданными похож на обмен любыми другими данными. Унификация обычно реализуется за счет скрытия нестандартного API за шлюзом API. Этот шаблон похож на шаблон «внутренние интерфейсы для внешних», описанный в главе 4

Помимо стремления к взаимодействию, вы должны четко позиционировать каждый инструмент и репозиторий, поскольку каждый из них ориентирован на отдельную область метаданных.

Хранилища метаданных

Области метаданных могут показаться сложными для понимания, поэтому далее я представлю самые важные функции и репозитории. Многие из этих возможностей метаданных не изолированы и тесно переплетаются друг с другом. Они также обычно сочетают самостоятельно созданные решения и готовые возможности.

- *Репозиторий приложений и баз данных.* Хранит информацию об именах приложений, настройках, базах данных, различных поставщиках, платформах, местоположениях, средах, мониторинге и состоянии работоспособности. Кластеризация различных приложений или их компонентов также позволяет узнать, какие бизнес-возможности они реализуют или в каком (ограниченном) контексте они используются.
- *Репозиторий списков золотых источников (List of Golden Sources, LoGS).* Хранит информацию о золотых источниках: уникальных наборах данных, содержащих золотые записи. Обычно эта информация дополняется сведениями о владельцах и наблюдателях, классификации, отношениях с основными и справочными данными предприятия и т. д.
- *Репозиторий контрактов на совместное использование и поставку данных.* Используется для регистрации и поддержки контрактов и ролей, связанных с предоставлением и потреблением данных. Эти контракты содержат общую информацию (происхождение), спецификации проверки данных, предоставленные данные (модель), архивирование, безопасность и качество данных. Эта функция используется также для копирования данных из одной среды в другую.
- *Репозитории для концептуальных, логических и физических моделей данных.* Собирают информацию о моделях данных и управляют ими. Концептуальные модели содержат модели отношений сущностей (entity relationship, ER), RDF и OWL. Логические модели содержат диаграммы единого языка моделирования (unified modeling language, UML), физические модели содержат языки определения данных, диаграммы отношений объектов, пояснительные диаграммы и/или диаграммы XML.
- *Репозитории для сообщений и моделей интерфейсов.* Модели сообщений и интерфейсов немного отличаются от моделей данных, потому что они обычно представляют собой абстракцию, созданную для обмена данными. В сервис-ориентированной (service-oriented architecture, SOA) и микросервисной архитектуре (microservices architecture, MSA) довольно распространено обслуживание моделей интерфейса: например, типов форматов объектов SOAP, XML Schema Definition Language (XSD) или JSON. В рамках потоковой передачи часто используются репозитории схем Avro. Для пакетов

обычно применяются объекты JSON или XML для описания интерфейса и типов объектов данных.

- *Репозитории для краудсорсингового контекста.* Эти репозитории содержат метаданные, которые делают контекст краудсорсинга более понятным. Например, дополнительные определения, описания, комментарии, оценки, а также вопросы и ответы.
- *Репозиторий с информацией о происхождении.* Служит для фиксации и отображения ETL, логики преобразования или потока данных. Обычно дополняется возможностью создания или визуализации отношений «модель — модель» на уровне атрибутов, что позволяет получать точное представление о происхождении и перемещении данных.
- *Репозиторий политик безопасности.* Используется для поддержки безопасности метаданных (функций безопасности, правил доступа и политик), чтобы обеспечить доступ к данным на уровне атрибутов. Репозиторий политик безопасности должен иметь тесную связь с репозиторием соглашений о совместном использовании данных, так как он определяет (и может отменять), что пользователям доступно или недоступно. Его можно применять для сбора информации об использовании, статистики и мониторинга, чтобы еще больше отточить элементы управления.
- *Репозиторий качества данных.* В этом репозитории фиксируются результаты применения правил качества данных, выполняемых при профилировании или сканировании источников данных. Он может содержать предложения по обнаружению взаимосвязей, расширению или удалению дубликатов.
- *Репозиторий классификации данных.* Используется для установки центральной и предметной классификаций данных, таких как конфиденциальность и этика, а также классификаций соответствия, персональных данных, нормативных требований, документов или управления жизненным циклом. Списки классификации используются во многих системах. Распространение и координация этих метаданных обычно осуществляется через системы управления справочными и основными данными.

Все репозитории метаданных, которые находятся на уровне метаданных, являются строительными блоками архитектуры для перспективного управления данными. Они сгруппированы вместе, потому что используются многократно и позволяют другим компонентам архитектуры работать вместе.

После надлежащего объединения центральный уровень метаданных будет выглядеть так, как показано на рис. 10.5. Локальные репозитории метаданных расположены вверху внутри центрального уровня метаданных, который стандартизируется путем повторного использования шаблонов интеграции уровня данных.

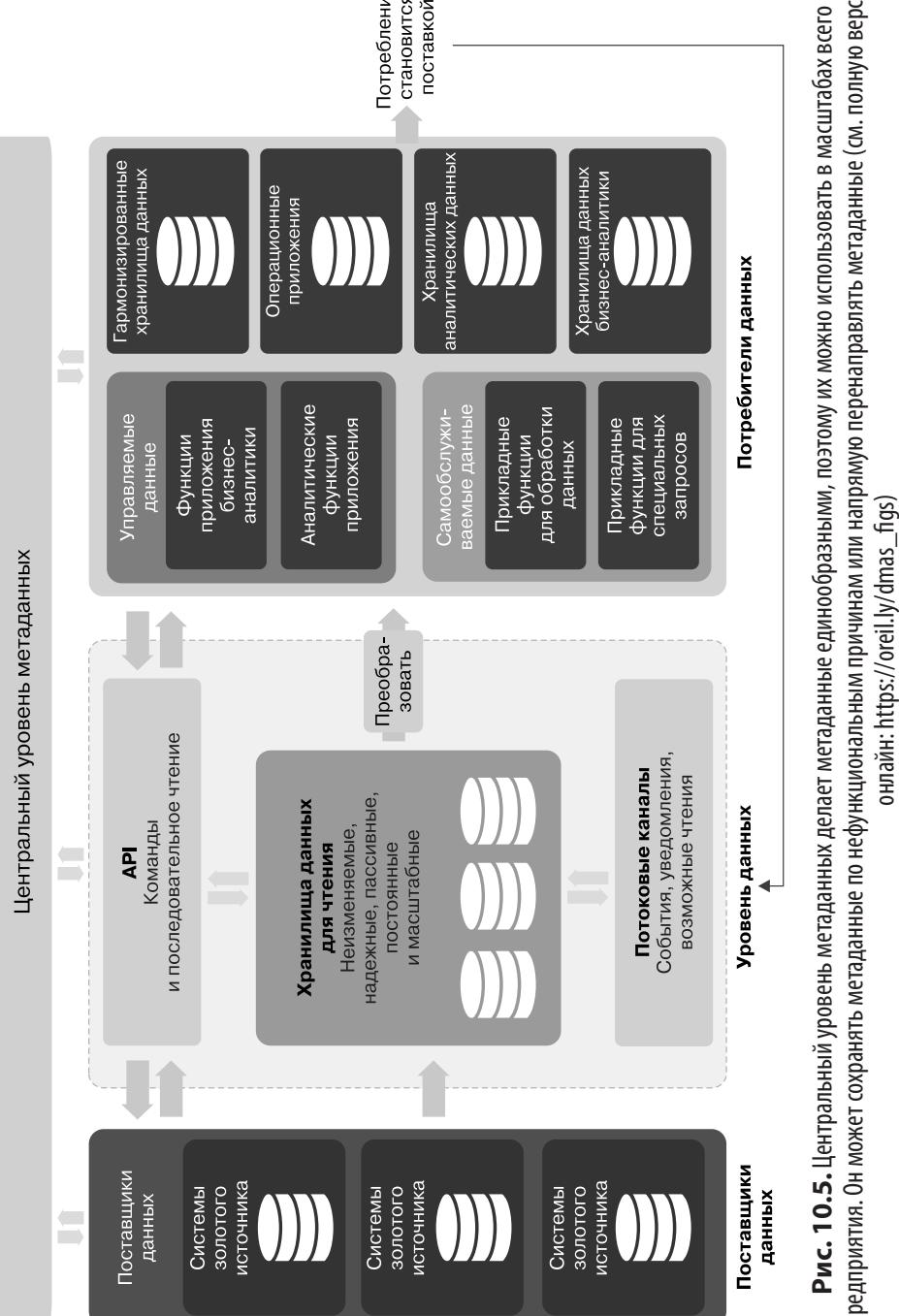


Рис. 10.5. Центральный уровень метаданных делает метаданные единообразными, поэтому их можно использовать в масштабах всего предприятия. Он может сохранять метаданные по нефункциональным причинам или напрямую перенаправлять метаданные (см. полную версию онлайн: https://oiel.ly/dmas_figs)

Площадка для быстрого доступа к авторизованным данным

В отрасли наблюдается растущая тенденция приближать данные к аналитикам и ученым с помощью порталов и площадок, которые интенсивно используют базовые метаданные. В этой сфере Airbnb (<https://oreil.ly/BiDP0>), Uber (<https://oreil.ly/kCdAX>), LinkedIn (<https://oreil.ly/hkBu6>), Netflix (<https://oreil.ly/0A4KN>), Lyft (<https://oreil.ly/XCIUT>) и другие компании разработали несколько решений. Все они стремятся к одной цели: демократизировать данные и дать сотрудникам возможность находить и использовать интересующие их данные.

Чтобы демократизировать данные через архитектуру, требуются четыре дополнительных архитектурных блока.

- *Портал самообслуживания* — позволяет поставщикам и потребителям данных совместно работать над выбором данных, которые можно сделать доступными. Этот портал должен предоставлять расширенные возможности поиска, позволяя пользователям выполнять поиск по ключевым словам, бизнес-терминам и фразам на естественных языках.
- *Самообслуживание* — относится к набору инструментов для автоматизации развертывания и предоставления общих моделей потребления, таких как бизнес-аналитика, обработка и исследование данных.
- *Панели мониторинга* — сообщают об общем состоянии работоспособности всех интерфейсов, потоков данных, подготовленных компонентов, центральных инструментов и т. д.
- *Интеллектуальные сервисы* — помогают автоматически прогнозировать, что пользователи будут искать, а также разумно собирать и готовить данные.

Сторона сотрудничества при демократизации данных часто связана с каталогами данных или полностью подконтрольными службами управления метаданными для облегчения поиска и обнаружения метаданных. Я рекомендую внимательно оценить потребность в каталогах коммерческих данных. Хорошо известные решения в этой области — Informatica EDC (<https://oreil.ly/FLT0s>), Lumada (<https://oreil.ly/mP1t6>), Collibra (<https://oreil.ly/GTY3y>) и Alation (https://oreil.ly/e_otw).

Крупные поставщики облачных услуг также предлагают свои решения. Это Azure Data Catalog (<https://oreil.ly/mP1t6>), AWS Glue Data Catalog (<https://oreil.ly/TnJi6>) и Google Data Catalog (<https://oreil.ly/jZVIZ>).

Время, необходимое для создания каталога данных, зависит от количества и разнообразия облачных платформ, баз данных, механизмов интеграции и используемых инструментов профилирования качества данных. Мой опыт показывает, что возможности, предлагаемые разными поставщиками, в основном зависят от области и тесно связаны с другими возможностями того же поставщика. Это может вызвать противоречия, так как каждая организация обычно имеет много областей, таких как собственные уникальные процессы, внутренний язык и терминология и основные направления.

Самостоятельное создание каталога данных, который объединяет информацию об их происхождении и качестве, обнаруживаемые источники и легкодоступные инструменты, — сложная задача. Поэтому я рекомендую использовать некоторые важные компоненты от коммерческих поставщиков, чтобы ускорить процесс. Важно, чтобы эти компоненты имели возможность подключаться к более широкой стратегии управления метаданными предприятия. Избегайте ловушки, связанной с использованием этих коммерческих инструментов в качестве основной серверной системы для всех метаданных. Постарайтесь оставаться максимально изолированными. В качестве альтернативы вы можете посмотреть на платформы с открытым исходным кодом, такие как DKAN (<https://getdkan.org/>) и CKAN (<https://ckan.org/>).

Перед тем как вы сможете разработать архитектуру рынка данных с самообслуживанием, управление метаданными должно достичь высокой организационной зрелости. Когда данные и метаданные рассредоточены, изолированы и не могут взаимодействовать друг с другом, невозможно обеспечить единое представление о том, какие данные доступны. Прочитав статьи крупных технологических компаний, вы быстро поймете, что все они либо консолидируют свои метаданные, либо унифицируют все свои API с помощью слоя обертки.

Если вы хотите создать внутреннюю площадку — при условии, что используете корпоративный каталог в качестве серверной части для большинства ваших метаданных, — архитектура должна выглядеть так, как показано на рис. 10.6.

Сама площадка обычно реализуется как тонкий слой оркестрации с привлекательным внешним видом. Он использует базовые репозитории метаданных, которые могут быть смесью собственных решений и готовых приложений. В зависимости от желаемой степени интеллектуальности вы можете добавить в реализацию площадки дополнительные аналитические возможности и возможности доступа к ресурсам.

Площадку с возможностями самообслуживания проще создать в облаке, чем в локальной среде, потому что многие поставщики облачных услуг удовлетворяют эти требования к самообслуживанию с самого начала.

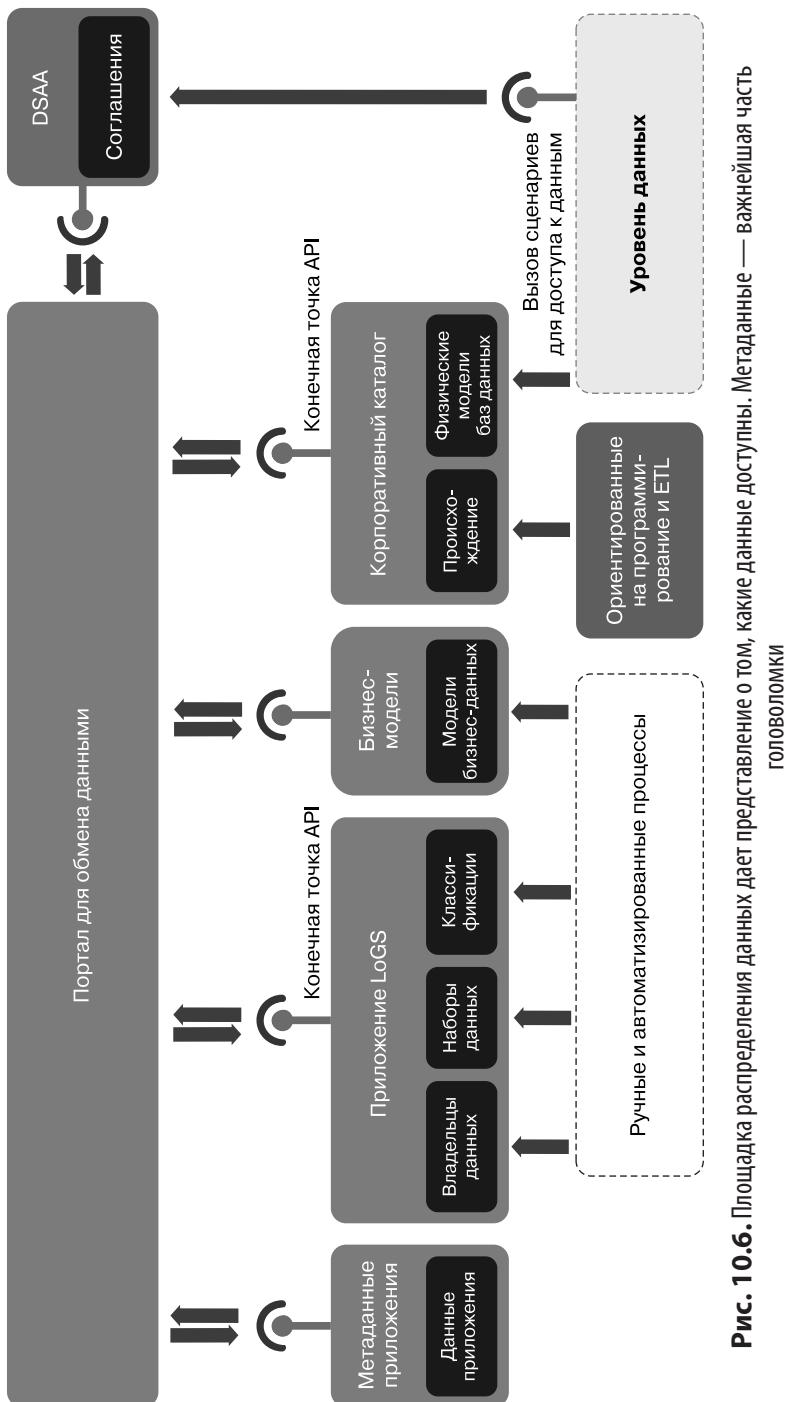


Рис. 10.6. Площадка распределения данных дает представление о том, какие данные доступны. Метаданные — важнейшая часть головоломки

Инструменты и API, предоставляемые поставщиком облачных услуг, можно использовать сразу же, без дополнительной настройки, и они легко интегрируются с механизмами самообслуживания, что показывают их инвестиции в 2019 году.

- Salesforce (<https://oreil.ly/0FZKK>) купила компанию по визуализации данных Tableau за 15,7 млрд долларов.
- Google объявила, что объединится с платформой Looker (<https://oreil.ly/a8wbx>) в рамках сделки на 2,6 млрд долларов.
- Microsoft приобрела BlueTalon (<https://oreil.ly/m0LOE>), компанию по обеспечению конфиденциальности и управления данными.

Облако, конечно, упростит внедрение самообслуживания, но я настоятельно рекомендую оценить, какие инструменты бизнес-аналитики и аналитики вы хотите предоставить. Самообслуживание и автоматическое выделение ресурсов сложно обеспечить, если у вас слишком много инструментов.

Самообслуживание может как создать, так и разрушить вашу новую архитектуру. Если пользователям будет сложно использовать архитектуру, все развалится. Создайте ее как серию небольших интуитивно понятных экосистем. Каждый архитектурный блок должен управляться и предоставляться как «повторно используемые сервисы», которые работают вместе, чтобы обеспечить полноценный опыт. Хорошая документация по API, отличная интеграция, ясность и простота использования процесса — вот ключевые признаки успешной реализации в масштабах компании. Пользователи склонны идти по пути наименьшего сопротивления. Если, например, выяснить происхождение данных сложно, но новая архитектура удовлетворяет эту потребность простым и удобным для пользователя способом, то пользователи с меньшей вероятностью будут пытаться определить ее самостоятельно. Чем проще использовать новую архитектуру, тем больше пользователей ее примут.

Итоги главы

Метаданные связывают все воедино. Проверка целостности и качества данных, их маршрутизация или копирование в новое место, преобразование данных и знание их значения — все это выполняется с помощью метаданных. Они необходимы и для демократизации данных через порталы самообслуживания.

Метаданные тесно связаны с дисциплинами качества данных, управления и интеграции и плотно взаимодействуют с архитектурой предприятия. Установите четкие границы владения данными и метаданными и следуйте принципам взаимодействия метаданных, чтобы обеспечить доставку и сбор метаданных высо-

кого качества, а также эффективное управление ими. Ваш отдел архитектуры предприятия должен принимать непосредственное участие в создании проекта архитектуры и управлении ею.

Управление метаданными — это не только большие данные, бизнес-анализ и аналитика. Интеграция метаданных также тесно связана с интеграцией приложений, их эксплуатационными и транзакционными аспектами. Ориентация на сервисы, микросервисы и DevOps, включая непрерывную интеграцию и развертывание, также является частью управления метаданными. Во всех этих областях группа управления данными должна решить, какие метаданные должны быть доступны централизованно.

Создание рынка данных — это не только метаданные. Оно также касается структуры, культуры и людей. Культурные аспекты требуют менее жесткого управления. Они требуют доверять пользователям, обучать людей и работать над осведомленностью. Эту деятельность не следует недооценивать. Ваши пользователи — ценные ресурсы, ведь они владеют определенной частью ландшафта данных или используют ее. Более эффективное взаимодействие с пользователями повышает эффективность знаний и применения данных.

ГЛАВА 11

Заключение

Ландшафты корпоративных данных нового поколения будут организованы совершенно иначе. Как вы узнали из этой книги, данные в ближайшие годы будут становиться все более распределенными, как и права собственности на них. Предприятиям все еще будут вынуждены покидать свою зону комфорта при управлении корпоративными моделями данных.

Масштабируемая архитектура, о которой шла речь в этой книге, имеет большое значение. Это новаторский подход, выходящий за рамки традиционного мышления. Он отличается универсальной архитектурой интеграции, которая содержит множество современных шаблонов, чтобы удовлетворить постоянно расширяющийся спектр нужд потребителей данных. Модели и принципы метаданных, описанные в этой книге, были разработаны для согласования способов распределения и интеграции данных. Три разные архитектуры интеграции нацелены на различные потребности интеграции, а метаданные все это объединяют.

Мелкие аспекты управления и безопасности делают масштабируемую архитектуру устойчивой и перспективной. В ближайшие годы в организациях, которые ставят данные в центр своего бизнеса, произойдет смена парадигмы. Это движение будет поддерживаться другой тенденцией: переходом от владения на основе приложений к владению на основе данных, что требует нового типа управления и других целевых операционных моделей, таких как DataOps и MLOps. Потребуется и другое мышление, включая продуктовое¹. Необходимо, чтобы сферы интеграции приложений, архитектуры ПО и управления данными были намного ближе друг к другу. Крупные предприятия будут вынуждены совмещать практику с очень разносторонними профессионалами во многих дисциплинах. Масштабируемая архитектура учитывает все эти разработки.

¹ При продуктовом мышлении (<https://oreil.ly/R8Fjq>) вы сосредотачиваетесь на реальных проблемах с целью создания хорошего пользовательского опыта. Оно включает в себя все, что приходит на ум, когда вы думаете об улучшении продукта: проще, красивее и удобнее в использовании.

Из этой книги вы узнали и о новых способах превращения данных в ценность. Помните о правиле 80/20: в большинстве случаев 80 % требований могут быть покрыты 20 % согласованных данных. Оставшиеся 20 % требований — самые сложные. Они будут связаны с задачами, характерными для уникальных вариантов использования. Тщательно проводя границы, вы можете полностью раскрыть потенциал своих данных. Имейте в виду, что от хранилищ данных никогда не требовалось обеспечить все аналитические варианты использования — только самые стабильные. Здесь также пригодится дисциплина управления мастер-данными (MDM). Если они текучие и быстроменяющиеся, то разбейте их на более мелкие части и оставьте их управление за предметными областями. Если они стабильны и достаточно зрелые, то рассмотрите возможность использования MDM.

В будущем, как вы узнали из главы 10, производителям придется заполнить много пробелов. Начнут появляться системы, готовые к работе с данными, и модели сервисов, способные обеспечить свободный и простой доступ к потребляемым данным. Я надеюсь, что в ближайшие годы мы увидим исправление недостатков метаданных. Более того, семантическое моделирование и практика интеграции данных начнут усиливать друг друга. Этот сдвиг ставит технические данные на уровень выше, ближе к бизнесу. Искусственный интеллект и машинное обучение тоже найдут свое применение во всех областях управления данными. Это сделает предприятия более эффективными.

Пройти через этот цифровой вихрь будет нелегко. Вам нужно выйти из зоны комфорта и позволить нескольким контекстам сосуществовать. Вы столкнетесь с сопротивлением со стороны областей и команд с традиционными инженерами данных. Вас могут заставить изменить организационную структуру. Важно убедить ИТ-команды в необходимости создания уровней инфраструктуры самообслуживания. Нужно разделить бункеры знаний и группы центральной платформы. Бизнес-пользователи должны нести ответственность за владение данными и управление ими. Команды безопасности должны знать о преимуществах самообслуживания и необходимости этики данных. Нужно назначить новые роли. Реализация вашего видения тоже требует выбора правильной модели доставки.

Модель доставки

Есть разные операционные модели и подходы к масштабной эксплуатации архитектуры. Все они имеют свои преимущества и недостатки. Ваш подход будет зависеть от размера и зрелости организации. Не забывайте, что управление и организационная стабильность являются важными факторами принятия этого решения. В каждой модели есть две постоянные стороны: команды предметных областей и команда центральной платформы.

- *Команды предметных областей* – это наследие предметно-ориентированного подхода к логической группировке частей организации. Каждая область должна состоять из владельца продукта (владельца данных), владельца приложения, разработчиков программного обеспечения и инженеров по обработке данных. Области должны четко сообщать друг другу о своих требованиях. В зависимости от размера организации и уровня технических знаний группы могут различаться по уровню автономии. Области несут ответственность за поддержание своих конвейеров данных и доставку или использование правильных наборов данных, но есть и исключения.
- *Команда центральной платформы* – предоставляет общую инфраструктуру со всеми необходимыми возможностями. Она должна позволять другим областям предоставлять и потреблять данные в режиме самообслуживания. Ожидается, что команда разработчиков платформы должна исключить выполнение любой повторной работы, насколько это возможно. Предполагается, что команда предоставит передовые технологии, поддержку, возможности непрерывной доставки и интеграции, возможности автоматического тестирования, инструменты поддержки конвейеров, таких как ETL. Ей стоит оставаться независимой от предметной области. Центральной команде нельзя принимать или создавать бизнес-логику либо приложения для других команд предметных областей.

В следующих подразделах я обрисую два разных подхода. В каждом из них команды областей и команда центральной платформы должны играть свою роль.

Полностью децентрализованный подход

Полностью децентрализованный подход делает области ответственными и автономными. Он основан на идее о том, что все области обладают достаточными знаниями о данных, архитектуре программного обеспечения и интеграции.

В этой модели важно предоставить широкий спектр возможностей, полностью управляемых и обслуживаемых самостоятельно. Сюда могут входить инструменты интеграции, такие как ETL, оркестрация, API и инициализация событий, регистрация схемы, порталы регистрации метаданных, мониторинг, управление идентификацией и доступом, аналитика и т. д. Эти возможности могут быть предоставлены централизованно и развернуты в модели звездообразной топологии сети, как описано в главе 6.

Очень важно, чтобы возможности было легко применять, поэтому нужно поддерживать дополнительную онлайн-документацию, реализации предопределенных сценариев, примеры кода, шаблоны CI/CD и т. д. В конце мы должны обеспечить значительную автоматизацию процессов и сбор метаданных. Как видите, для достижения цели нужно тесно сотрудничать с профессионалами, имеющими

разный опыт в разных сферах – от организации инфраструктуры до управления данными, от облачных технологий до архитектуры программного обеспечения.

Основное преимущество полностью децентрализованного подхода – масштабируемость. Потому что для создания решений нужны лишь ограниченные возможности центральной команды. Скорость доставки из источников зависит от осведомленности и способности предметных областей. Когда команда очень зрелая, ее участникам может быть предоставлена большая автономия и они могут выбирать собственные инструменты. Пока они интегрируются с объединенной архитектурой, все должно находиться под полным контролем. Такая модель больше всего подходит для предприятий с большим количеством областей и выделенных ИТ-отделов.

Частично децентрализованный подход

При *частично децентрализованном подходе* модель немного меняется, чтобы обеспечить командам областей пул централизованно управляемых ресурсов. В этой модели команда центральной платформы предоставляет знания или дополнительную пару рук, когда это необходимо.

Я видел, что эта модель особенно хорошо работает на небольших предприятиях с менее зрелыми командами предметных областей и в случаях, когда предпочтение отдается строгой стандартизации продуктов. В этой модели области по-прежнему заботятся о создании и разработке решений, но используют центральные инструменты и передовые практики. По логике вещей в этой модели больше возможностей диктуется централизованно. Это означает меньшее количество возможностей для выбора инструментов и методов.

Главное преимущество этой модели в том, что обычная практика позволяет предметным областям взаимодействовать и обмениваться знаниями. Недостаток – более тесная связь с центральными инструментами. Если по какой-либо причине центральные механизмы выйдут из строя или отключатся, то областям придется подождать.

Структурирование команд

Что бы вы ни выбрали, я рекомендую провести четкую грань между командами технологических инженеров и группами консультантов по управлению данными.

Вся инфраструктура, не зависящая от предметной области, должна представляться командами платформы, как показано на рис. 11.1: разработчиками, которые создают базовую инфраструктуру и шаблоны, позволяющие группам предметных областей взаимодействовать друг с другом. К ним относятся шаблоны приема данных, шаблоны потребления, регистрация метаданных,

а также обеспечение безопасности и политики. На большом предприятии таких команд будет несколько. Каждая должна быть организована вокруг основной возможности. В каждой команде должен быть выделенный владелец продукта и архитектор.



Рис. 11.1. Рекомендации по составу команды

Рядом с командой платформы должна быть создана группа консультантов по управлению данными для гарантии того, что все возможности управления данными используются как нужно. Желательно, чтобы группы консультантов по управлению данными не зависели от технологий. Эти группы, в состав которых входят консультанты по данным, должны сосредоточиться на взаимодействии и опыте пользователей: подключении, консультировании и поддержке владельцев данных; регистрации метаданных; мониторинге использования возможностей потребления и т. д.

Помимо команд технологического проектирования и управления данными, должна иметься команда управления, занимающаяся согласованием незавершенных работ, решением проблем и взаимодействием с сообществом. Эта команда должна также организовать демонстрации для более широкой организации.

Стратегия InnerSource

Часть хорошей структуры и целевой операционной модели — эффективное развертывание ресурсов и обеспечение роста общей программы обработки данных.

Я рекомендую взглянуть на операционные модели сообщества с открытым исходным кодом, в частности на модель *InnerSource*¹.

InnerSource позволяет разработчикам программного обеспечения вносить свой вклад в усилия других команд, способствуя прозрачности и открытости. Эта прозрачность включает стратегию, общие списки дел, цели и планирование. Когда все открыто, люди могут лучше видеть зависимости и определять, какие работы необходимо выполнить срочно. В этом случае разъясняется, какие ресурсы доступны в настоящее время и как их можно использовать более эффективно. Например, если что-то сдерживает все остальные команды, вы можете временно выделить больше ресурсов для решения проблемы. Но на этом модель не останавливается.

Стратегия *InnerSource* предназначена еще и для создания компонентов и шаблонов, которые можно использовать повторно для всего сообщества. Поэтому, если одной команде требуется конкретный шаблон, его следует спроектировать так, чтобы задействовать все остальные команды. Пример — развертывание набора API чтения поверх RDS. Одна команда могла выполнять всю тяжелую работу только за себя, но эта же команда могла также объединиться с командой разработчиков платформы для централизованной работы. Такой тип координации требует нового типа лидерства и поведения.

Культура

Построение организации, основанной на данных, и необходимый переход потребуют от вас создания новой культуры. Вам нужно начать нанимать «специалистов по данным», «сторонников данных» и «вдохновляющих лидеров». Пригласите специалистов в области данных, обладающих знаниями в области ИТ и бизнеса. Еще вам понадобятся «евангелисты», чтобы наладить отношения с руководством высшего уровня и найти хороших спонсоров. Для изменения организационной культуры следует обосновать необходимость изменений. Начните готовить свою организацию к переменам в мире данных и влиянию этих перемен на управление данными. Это означает активную агитацию и «спасение душ», а также готовность стоять на своем, когда дела пойдут в неверном направлении.

Предметное мышление и DataOps требуют определенного склада ума. Все участники должны начать работать вместе, от создателей и поставщиков данных

¹ В книге Данезе Купера (Danese Cooper) и Клааса-Яна Стола (Klaas-Jan Stol) *Adopting InnerSource: Principles and Case Studies* (O'Reilly, 2018) представлены тематические исследования и рекомендации по внедрению *InnerSource*.

до пользователей, которые создают отчеты и аналитические модели. Им нужно подумать об автоматизации и самообслуживании. Для достижения успеха вам нужно сделать данные удобными и эффективными, позволить бизнес-пользователям скрыться от корпоративного представления — это означает довериться сообществу и поддерживать его с помощью соответствующего обучения и дополнительных инструментов.

Выбор технологий

После завоевания сердец и умов заинтересованных сторон важно начать строить и показывать результаты. Начальные проекты не должны быть слишком большими. Начните с малого и быстро покажите результаты, чтобы заинтересованные стороны сразу увидели преимущества использования данных. Это поможет заручиться поддержкой руководителей высшего звена. Как только вы продвинетесь дальше, ваш проект должен стать образцом для подражания, что позволит вам свернуть или скорректировать другие инициативы.

Что касается технологий, то проверьте множество вариантов. Крупные предприятия обычно имеют разнообразные системы, платформы данных, поставщиков, предложения SaaS и т. д. Для сбора всех данных и их объединения нужно большое количество интеграционных решений. Здесь тоже важно начать с малого и создать начальную основу с инфраструктурой для сбора данных. Стандартизируйте возможности и убедитесь, что все придерживаются этих стандартов.

Дальше выберите технологии, позволяющие объединять и распространять данные в нужном масштабе. Здесь я советую положиться на крупных поставщиков и обратить внимание на облако. Проектируйте все возможности, следя за моделями «как услуга» и разработайте план метаданных. Помните, что эти связи должны скреплять вашу архитектуру.

По мере продвижения вперед нужно соединить все разные варианты использования с вашей архитектурой. В этом случае тоже не следует торопиться: выберите варианты использования, которые легко реализовать и которые принесут наибольшую пользу вашей организации. Затем увеличивайте масштаб с помощью автоматизации и вводите самообслуживание. Подключите циклы обратной связи, присоедините к своей архитектуре решения MDM и качества данных.

Наконец, нужно создать совместный уровень. Я считаю, что он должен быть основан на графах, которые, в свою очередь, основаны на онтологиях. Это позволит областям расширять корпоративный граф, давая им возможность совместно использовать гораздо больше контекста, который затем можно вне-

дрянь полностью автоматически. Интеллектуальные возможности потребления, которые могут это использовать, — вишненка на торте.

Рассмотрите возможность построения вашей архитектуры как открытой платформы для совместной работы. В широкой экосистеме с открытыми данными, финансово-технологическими компаниями, поставщиками SaaS, а также внешними поставщиками данных и потребителями ваша платформа должна разрабатываться с учетом открытости. Чтобы монетизировать данные в больших масштабах, вы должны изучить варианты предоставления ваших информационных ресурсов другим компаниям. Вы можете стать брокером данных (<https://oreil.ly/a08V4>)!

Монетизация данных дает дополнительные возможности. Данные должны поставляться как услуга для облегчения объединения внутренних и внешних источников. Все эти возможности связаны с шаблонами интеграции. Кроме того, вы можете предоставить аналитику как услугу, чтобы делиться ценной информацией из данных со сторонними партнерами, использующими платформу. Для рассылки этой информации в реальном времени можно положиться на шаблоны, описанные в главе 5.

Упадок традиционной архитектуры предприятия

Роль архитекторов предприятия изменится, так же как и сами архитектуры. Архитекторов нужно рассматривать как лидеров, которые могут направлять команды разработчиков и содействовать им в реализации архитектуры. Они должны быть прагматичными и реалистичными, но при этом определять перспективы и вдохновлять каждого следовать одной цели. Архитекторы должны дышать технологиями и преуспевать во многих областях, таких как безопасность, облачные технологии, архитектура ПО, интеграция и данные. Новые корпоративные архитекторы должны иметь очень глубокое понимание границ бизнеса и знать, как разделить их, пользуясь современными шаблонами интеграции.

Такое представление о роли архитектора, как стратега, высококвалифицированного инженера, хранителя — далеко от традиционного стереотипа архитектора, как человека, мыслящего только статическими объектами и диаграммами и находящегося глубоко внутри ИТ-отдела. Внедрение практики архитектуры предприятия (enterprise architecture, EA) требует знаний в самых разных областях деятельности. Давайте посмотрим, что отличает традиционных архитекторов.

Чертежи и схемы

Архитектура предприятия обычно выражается в виде формализованных моделей или диаграмм Visio. В эпоху динамичных предприятий это ремесло больше не масштабируется. В более современной практике модели и диаграммы должны строиться на основе базовой метамодели с использованием репозитория кода. Этот репозиторий должен постоянно дополняться информацией о моделях данных, стоимости, происхождении, новых приложениях, правах собственности и т. д. Поверх этого репозитория должны быть развернуты средства визуализации, показывающие структуру предприятия: его бизнес-возможности, отделы, приложения, базы данных и т. д.

Создание и визуализация этих современных артефактов очень похожи на то, что вы видели в главе 10. С помощью передовых технологий вы можете дополнить эту возможность стратегическими сценариями, прогнозами и разнообразными показателями.

Современные навыки

Изменятся и основные навыки архитекторов предприятий. В то время как традиционно архитекторов считают экспертами, в новую эру они должны овладеть комплексными приемами решения проблем, научиться критически и творчески мыслить.

Этот переход от эксперта к специалисту по решению проблем требует воспитания качеств лидера и новатора, что можно наблюдать в успешных консалтинговых фирмах. Это требует владения знаниями и умениями, такими как дизайн-мышление, прототипирование, концепции моделирования бизнеса, ассоциативные карты, проектирование взаимодействий с пользователями, и многими другими. Страйтесь определять реалистичные цели и тщательно управлять ожиданиями.

Умение слушать и общаться очень важно для нового архитектора. Не пытайтесь перегружать представителей бизнеса архитектурным жаргоном. Страйтесь быть напористыми, открытыми и прагматичными или найдите людей, обладающих этими качествами.

Контроль и управление

Развивая архитектуру своего предприятия, ищите баланс между долгосрочными целями и практичностью. Это может привести к отказу от базовых концепций

строительства архитектуры предприятия (таких как стандарт TOGAF платформы Open Group Architecture Framework), ведь их логика формализации давно устоявшихся и статических в будущем состояний не вписывается в новые миры DevOps и DataOps. Да, вам нужно нарисовать «общую картину», но вы также должны избавиться от закостенелого мышления. Современный архитектор должен стать лидером сообщества. Он должен взять на себя инициативу по определению минимально жизнеспособных продуктов, организуя встречи и обсуждения, а также транслировать потребности клиентов. Оставьте детали команде, но будьте авторитетным экспертом на случай, если что-то пойдет не так.

Послесловие

Хотя термин «архитектура предприятия» имеет негативные ассоциации, я твердо уверен, что его принцип работы от начальной идеи до окончательной реализации по-прежнему имеет решающее значение. Всегда будет необходимость набросать общую стратегию, разработать абстрактную концепцию, представить идею, активизировать ее и поддержать команды в исполнении этой стратегии. Такой процесс лежит в основе масштабируемой архитектуры, которая, в конце концов, началась с идеи.

Последний совет: не бойтесь! В чудесном мире данных можно найти много всего интересного. Эта книга — только начало.

Глоссарий

ACID

ACID расшифровывается как atomicity, consistency, isolation, durability – «атомарность, согласованность, изоляция, долговечность».

Атомарность гарантирует, что транзакция либо выполнится полностью, либо не будет выполнена вовсе. *Согласованность* гарантирует, что по завершении любой транзакции система будет находиться в допустимом состоянии. *Изоляция* гарантирует, что система выполняет только одно действие за раз. *Долговечность* означает, что после успешного завершения транзакции изменение становится постоянным.

Стандарты ACID обеспечивают бесперебойную работу и безопасное восстановление БД, потому что они достоверны даже в случае прерываний, ошибок, сбоев питания и т. д. В ACID транзакции рассматриваются как отдельные (атомарные) операции. Либо они выполняются все, либо не выполняется ни одна. Транзакции имеют только одно действительное состояние и изолированы на момент записи или обновления.

Apache Avro

Apache Avro – проект с открытым исходным кодом, который предоставляет поддержку сериализации и обмена данными для экосистем Apache Kafka и Apache Hadoop. В нем есть служебные программы, которые можно использовать для эффективной сериализации данных в файлы или сообщения. В его основе лежит репозиторий. В Avro данные всегда сопровождаются схемой. Он хранит определение данных в JSON. Apache Avro в настоящее время хорошо работает в экосистеме Hadoop (включая Apache Kafka).

Apache Thrift

Apache Thrift был разработан в Facebook в 2007 году. Это проект с открытым исходным кодом. Он использует широкий спектр языков и предлагает полный стек «клиент/сервер», с которым могут напрямую работать многие проекты. Он также использует IDL (язык определения интерфейса) для описания типов данных, который очень похож на JSON и легко читается.

BASE

Аббревиатура BASE была определена Эриком Брюером (Eric Brewer), который также известен формулировкой теоремы CAP. BASE означает «базовая доступность», «мягкое состояние», «согласованность со временем».

CQRS

Разделение ответственности команд и запросов (command and query responsibility segregation, CQRS) отделяет операции чтения данных (запросы), от операций, изменения данных (команды), с помощью отдельных интерфейсов.

ELT

Процесс извлечения, загрузки и преобразования (extract, load, transform, ELT) – это альтернатива процессу извлечения, преобразования и загрузки (extract, transform, load, ETL), используемая с реализациями базы данных или озерами данных. В отличие от ETL в моделях ELT данные не преобразуются при входе в озеро данных, а сохраняются в исходном необработанном формате.

ETL

ETL (extract, transform, load) – процесс извлечения, преобразования и загрузки данных.

HOLAP (гибридная оперативная аналитическая обработка)

Продукты HOLAP объединяют MOLAP (Multidimensional OLAP – многомерная оперативная аналитическая обработка) и ROLAP (Relational OLAP – реляционная оперативная аналитическая обработка), используя реляционную БД для большей части данных и отдельную многомерную БД для наиболее плотных данных, обычно составляющих небольшую часть данных.

MOLAP (многомерная оперативная аналитическая обработка)

Продукты MOLAP обеспечивают многомерный анализ данных, помещая их в структуру куба. Данные в этой модели сильно оптимизированы для максимальной производительности запросов.

MQTT

Транспорт телеметрии очереди сообщений (message queuing telemetry transport, MQTT) – это протокол обмена сообщениями на основе схемы публикации-подписки, которая занимает очень мало места, когда пропускная способность низкая. Это делает MQTT одним из наиболее часто используемых протоколов в проектах интернета вещей.

NewSQL

Базы данных NewSQL¹ пытаются предложить лучшее из обоих миров: масштабируемость и скорость NoSQL в сочетании с реляционной моделью данных и согласованностью транзакций ACID. В то время как NoSQL является хорошим выбором для обеспечения максимальной доступности, более быстрого времени отклика и согласованности в конечном итоге, NewSQL

¹ Термин NewSQL был введен Мэттом Аслеттом (Matt Aslett) для описания новой группы БД, которые разделяют большую часть функциональности традиционных реляционных баз данных SQL, но при этом предлагают некоторые преимущества технологий NoSQL.

при использовании распределенной архитектуры делает упор на согласованность, а не на доступность.

NoSQL

Карло Строцци (Carlo Strozzi) первоначально ввел термин *NoSQL*, чтобы заявить, что его легкая реляционная база данных с открытым исходным кодом не использует стандартный язык структурированных запросов (SQL). Десять лет спустя Йохан Оскарссон (Johan Oskarsson) снова использовал этот термин для описания нереляционных баз данных. Термин *NoSQL* может означать либо «Нет SQL», либо «Не только SQL».

OLAP

OLAP (online analytical processing – оперативная аналитическая обработка) – это технология, лежащая в основе многих приложений бизнес-аналитики (BI). OLAP – это мощная технология для обнаружения данных, включающая возможности неограниченного просмотра отчетов, сложных аналитических расчетов и прогнозирующего сценария «что, если» (бюджета, прогноза).

OLTP

OLTP (online transactional processing – оперативная обработка транзакций) – это категория обработки данных, которая направлена на задачи, ориентированные на транзакции. OLTP обычно включает в себя вставку, обновление и/или удаление небольших объемов данных в БД. OLTP в основном имеет дело со множеством транзакций, совершаемых большим количеством пользователей.

Pull-запрос

Поставщик данных ждет, пока потребитель обратится к нему для извлечения данных. После этого поставщик должен подтвердить запрос и начать доставку данных. В этой модели потребитель играет активную роль и может решать, когда начинать запрашивать данные.

Push-запрос

Как только поставщик данных обнаружит, что у него есть некоторые данные для отправки другой стороне, данные будут отправлены потребителю через конвейер передачи данных. Поставщик в этой модели играет активную роль и может определять, когда инициировать передачу. Потребитель должен будет обрабатывать полученные данные.

ROLAP (реляционная оперативная аналитическая обработка)

Продукты ROLAP работают в тесном взаимодействии с реляционными БД для поддержки OLAP. Часто структура схемы типа «звезды» используется для расширения и адаптации базовой структуры реляционной базы данных, чтобы она представляла себя как сервер OLAP.

Авторизация

Авторизация — это предоставление доступа к определенным частям данных. Предоставление доступа основано на ролях. Такие концепции, как *система контроля доступа*, действуют, опираясь на заданные интервалы времени, в течение которых токен безопасности остается действительным. Крупные компании обычно используют корпоративную технологию *Active Directory*, которая поддерживает назначение ролей пользователям. Эти роли обычно основаны на должности или корпоративном статусе.

Архив наборов данных

Архивные данные¹ не имеют отношения к приложению и не являются частью бизнес-процессов, но должны быть сохранены для соблюдения нормативных требований или в юридических целях. Архивные данные обычно сохраняются во вторичном, наименее дорогостоящем хранилище для нечастого обращения и обнаружения. Обычно приложения не могут напрямую обращаться к архивным данным без соблюдения дополнительных мер безопасности.

Архитектура будущего состояния

Архитектура будущего состояния — это схема, показывающая, как архитектура должна выглядеть в будущем. Она должна соответствовать видению бизнеса и создаваться в тесном сотрудничестве с командой ЕА.

Архитектура Инмона

По мнению Билла Инмона (Bill Inmon), любая часть данных, которая может оказаться полезной, должна храниться в единой универсальной модели данных, являющейся «единым источником истины» для предприятия. В этом источнике истины используется конструкция целочисленной БД с эффективным хранением, обычно это структура 3NF. Из этого источника истины выбираются (подмножества) конкретные проекты, варианты использования или отделы. Это выбранное подмножество, которое оптимизировано для производительности чтения варианта использования, называется витриной данных².

Архитектура Кимбалла

По мнению Ральфа Кимбалла (Ralph Kimball), хранилище должно представлять собой конгломерат или набор размерных данных, которые являются

¹ Крупные поставщики облачных услуг часто ссылаются на горячее, теплое и холодное хранилище. Хотя эти уровни хранения можно сопоставить с каждым этапом жизненного цикла данных, они не совпадают. Управление жизненным циклом данных использует точку зрения управления данными и является дисциплиной. Это не зависит от уровней хранения или доступа. Например, исторические и архивные данные могут быть неактуальными с точки зрения бизнеса, но должны храниться в хранилище одного и того же типа.

² Витрины данных не ограничиваются архитектурой Инмона, они используют и другие стили хранилищ данных.

копиями данных транзакций из исходных систем. Поскольку хранилище данных требуется для различных сценариев использования, Кимбалл создал концепцию «согласованных измерений»¹, которые являются ключевыми измерениями, общими для разных потребителей и используемыми ими.

Архитектура текущего состояния

Архитектурная схема, описывающая высокоуровневую архитектуру текущей организации, которая включает все бизнес-подразделения и общие технологические сервисы. В большинстве организаций за составление таких схем отвечает команда корпоративной архитектуры (EA).

Архитектурный блок (Architecture building block, ABB)

Составная часть модели архитектуры, описывающая не зависящий от продукта пакет функций общей модели², определенный для удовлетворения конкретного типа бизнес-потребностей в организации. Например, возможность регистрации клиентов, которая может потребоваться на предприятии.

Аутентификация

Аутентификация — это проверка прав доступа. Когда пользователь или процесс пытается войти в приложение, систему или БД, важно проверить его личность. Один из способов подтвердить личность — ввести имя пользователя и пароль; еще один вариант — ключ безопасности. Когда происходит аутентификация, весь обмен данными обычно шифруется, чтобы предотвратить кражу учетных данных аутентификации.

База данных

База данных — это часть программного обеспечения, используемого для организации данных, обычно хранящихся и доступных в электронном виде из компьютерной системы.

База данных на основе столбцов

Подобные БД, как и реляционные, используют таблицы, строки и столбцы. Разница в том, что имена и форматы столбцов более гибкие и могут меняться от строки к строке внутри одной и той же таблицы. Другое отличие состоит в том, что поля физически хранятся по-разному, столбец за столбцом, а не строка за строкой. Следовательно, производительность при выполнении тяжелых агрегатов выше, поскольку все поля определенного столбца физически хранятся вместе. Примеры: Cassandra, Google BigTable и HBase.

¹ Согласованные измерения — это измерения, которые имеют общее, согласованное значение в рамках более крупного предприятия.

² Часто возникает вопрос: зачем нам и архитектурные блоки (ABB), и строительные блоки решения (SBB)? Это необходимо для того, чтобы выявлять отклонения ли реализации (SBB) от архитектурных спецификаций (ABB).

Базы данных «ключ — значение»

Хранилища «ключ — значение» или базы данных «ключ — значение» не используют фиксированную или предопределенную структуру, а хранят данные в виде массивов. Эти типы БД работают, сопоставляя ключи со значениями, подобно словарю. Хранилища «ключ — значение» обычно используются для быстрого хранения и извлечения основной информации. Примерами являются Redis, MemcacheDB, Riak и Berkeley DB.

Базы данных, оптимизированные для поиска

Такие БД оптимизированы для обеспечения функций текстового поиска и индексирования документов. Эти функции позволяют реализовать поиск в реальном времени, когда пользователи вводят определенные термины. Примеры: Elasticsearch, Solr и Lucene.

Базы данных только для добавления

Добавляет данные в конец файла или базы данных. Попытки изменения или добавления данных в середину не имеют эффекта. Их преимущество в том, что запись в БД выполняется быстрее, а файл транзакций сохраняется. Самый большой недостаток состоит в том, что все изменения добавляются в виде новых записей, что влечет увеличение используемого объема хранилища.

Бизнес-аналитика

Бизнес-аналитика, по определению Gartner (<https://oreil.ly/Pydz>), — это «общий термин, который включает приложения, инфраструктуру и инструменты, а также передовые методы, которые обеспечивают доступ к информации и ее анализ для улучшения и оптимизации решений и производительности». Бизнес-аналитика началась с создания простых отчетов, но на более позднем этапе было введено *самообслуживание* с более продвинутыми возможностями, такими как обработка памяти и предсказуемые функции.

Бизнес-метаданные

Бизнес-метаданные используются для определения смысла и (поведенческого) бизнес-контекста. Примерами являются концептуальные модели данных, бизнес-глоссарии, таксономии данных, определения данных, классификации и информация о владении данными.

Блокчейн-хранилища

Блокчейн-хранилища обеспечивают доверие и прозрачность для совместного использования и распространения данных. Блокчейн группирует наборы данных в блоки с использованием технологии хеширования. Хеши пересыпаются между узлами, чтобы гарантировать, что данные не были скомпрометированы. Примеры: Hyperledger и Quorum.

Виртуализация данных

Виртуализация данных — это подход, который позволяет приложению извлекать данные и манипулировать ими, не требуя технических подробностей о данных, таких как их формат в источнике или местоположение, и позволяет создать единое представление общих данных (или любых других объектов). Некоторые поставщики баз данных предоставляют уровень запросов к базе данных (*виртуальный*), который также называется *уровнем виртуализации* данных. Этот уровень абстрагирует базу данных и оптимизирует данные для повышения производительности чтения. Еще одна причина для абстрагирования — перехват запросов для большей безопасности. Примером может служить Amazon Athena.

В памяти

Понятие *в памяти* (In-memory) относится к данным, которые полностью хранятся в основной памяти. Оперативная память работает быстрее, чем запись и чтение из файловой системы.

Выгрузка данных

В рамках управления данными их выгрузка — это процесс уменьшения объема данных за счет перемещения в другую систему. Анализ оставшихся данных будет быстрее, потому что обрабатывается меньше данных.

Гибридная транзакционная/аналитическая обработка

Некоторые системы сочетают OLTP и OLAP. Эти системы называются *гибридной транзакционной/аналитической обработкой*, как предложено компанией Gartner. Хотя эти системы снаружи выглядят как новые архитектуры, внутри все еще есть две БД. Одна из них предназначена для множества небольших транзакций с высокой долей обновлений, другая (находящаяся в памяти) обрабатывает сложные запросы для аналитических рабочих нагрузок.

Графовая база данных

Такие БД используют древовидные структуры (графы) с узлами и ребрами, соединяющими друг друга через отношения. Они часто используются для хранения отношений (социальных) между объектами. Некоторые БД на основе графов также поддерживают пространственные данные или типы геоданных. Данные определены в геометрическом пространстве, оптимизированном для геометрических объектов, таких как точки, линии и многоугольники. Примеры: Neo4J, Titan, Amazon Neptune и Giraph.

Данные

Любые значения из приложения, которые можно преобразовать в факты и в конечном итоге в информацию¹.

¹ Факты также могут быть взяты из документов, а затем сохранены в виде электронных файлов. Это называется управлением записями.

Данные в движении

Считается, что данные в движении немного отличаются от данных в состоянии покоя, поскольку данные перемещаются в пределах самой инфраструктуры приложения, а не передаются за пределы сервера и не покидают физическую инфраструктуру или оборудование. Иногда серверы имеют прямое частное соединение и совместно используют базовое хранилище. Данные можно копировать через общую (сетевую) среду хранения, а не через традиционную сеть. То же верно для данных в пути.

Данные в пути

Данные в пути – это данные, которые активно перемещаются из одного места в другое, например, через интернет или через частную сеть.

Данные в состоянии покоя

После того как данные скопированы и сохранены, они обычно находятся в базе данных или файле. Мы называем такие данные «данными в состоянии покоя».

Данные транзакции

Данные транзакции – это информация о событиях и транзакциях. Она тесно связана с основными и справочными данными, но не имеет специальной дисциплины.

Документоориентированная система управления базами данных

Хранилища документов или БД, ориентированные на документы, имеют сходство с хранилищами пар «ключ – значение». Они имеют расширенную реализацию с более сложными функциями, организуют документы в коллекции и поддерживают глубоко вложенные и связанные сложные структуры данных. Примеры: MongoDB, Amazon DocumentDB и CouchDB.

Допустимость

Степень допустимости данных с точки зрения синтаксиса (формат, тип, диапазон значений). Телефонные номера могут храниться в одной таблице в форматах: +44123456789 и 0044123456789.

Журнал транзакций

Журнал транзакций хранит сведения обо всех транзакциях (модификациях в базе данных) в отдельном файле или БД. В случае сбоя журнал транзакций можно использовать для проверки, извлечения или восстановления данных.

Завершенность (полнота)

Степень полноты доставленных или сохраненных данных и отсутствия пропущенных значений. Примеры: пустые или отсутствующие записи.

Захват изменения данных

Захват изменения данных (change data capture, CDC) – это набор шаблонов разработки программного обеспечения, используемых для определения

(и отслеживания) изменений в данных, чтобы можно было предпринять соответствующие действия. CDC часто использует журнал транзакций базы данных фиксации различий, хотя он также может напрямую запрашивать базу данных.

Золотая запись

Золотой источник и золотой набор данных имеют сходство с тем, что люди называют золотыми записями. Tech Target (<https://oreil.ly/cAxs->) определяет золотую запись как «единственную версию истины», где *истина* понимается как ссылка, к которой пользователи данных могут обращаться, когда хотят убедиться, что у них есть правильная версия части информации. Золотая запись включает в себя *все* данные в каждой системе записей (system of records, SOR) в конкретной организации».

Индекс базы данных

Индекс, или индекс базы данных, — это структура данных, используемая для быстрого поиска и доступа к данным в таблице базы данных.

Интеграция данных

Интеграция данных — это действия, методы и инструменты, необходимые для консолидации и согласования данных из разных (нескольких) источников в едином представлении. Процессы извлечения, преобразования и загрузки (ETL) являются частью этой дисциплины.

Используемые данные

Данные, которые находятся в памяти в момент использования или обработки внутри приложения.

Исторические данные

Исторические данные — это предыдущие состояния текущих данных. Исторические данные не имеют значения, но должны сохраняться для бизнес-процессов или случайного доступа.

Каноническая модель

Каноническая модель — это шаблон проектирования для обмена данными между различными форматами данных, независимо от выбранной технологии. Термин «канонический» в моделировании данных просто означает «один». Обычно, когда архитекторы говорят о канонических моделях данных, они имеют в виду, что существует один канонический язык (как правило) для всего предприятия.

Компонент приложения

Модульная, развертываемая и заменяемая часть системы, которая инкапсулирует ее данные и функции, предоставляя их через набор интерфейсов. Например, клиентский интерфейс системы CRM, который обрабатывает взаимодействие с пользователем.

Корпоративная модель

Корпоративная модель данных — это единый автономный унифицированный артефакт, который описывает все объекты и атрибуты данных, а также их взаимосвязи в рамках предприятия. Часто эта модель сочетается со стремлением объединить все данные в корпоративном хранилище данных (EDW).

Корпоративное хранилище данных

Корпоративное хранилище данных (enterprise data warehouse, EDW) — это БД или набор БД, которая централизует данные из нескольких источников и приложений и делает их доступными для аналитики и использования во всей организации.

Кубы

Кубы, также известные как кубы OLAP, — предварительно обработанные и обобщенные наборы данных, которые сильно сокращают время запроса. Оперативная аналитическая обработка (online analytical processing, OLAP) относится к специализированным системам или инструментам, которые делают данные легкодоступными для принятия аналитических решений. Кубы OLAP — это логические структуры, определенные в метаданных. Многомерные выражения (multidimensional expressions, MDX) — популярный язык запросов на основе метаданных для запросов к кубам OLAP.

Кэширование

Кэширование относится к размещению часто используемых записей в области (памяти или оптимизированном хранилище) для более быстрого доступа.

Происхождение данных

Происхождение данных определяет, откуда произошли данные и куда перемещаются с течением времени.

Маскировка

Маскировка — это процесс скрытия данных. Маскировка может включать удаление, перемешивание или замену значений. Типичный пример — замена последних 12 цифр 16-значного номера кредитной карты.

Материализованное представление

Оптимизированный объект базы данных, содержащий результаты запроса. Материализованные представления можно рассматривать как форму кэширования.

Метаданные

Метаданные описывают сами данные. Этот термин часто используется в отношении цифровых носителей, но в современном мире он играет жизненно важную роль в общей стратегии обработки данных и архитектурном

проектировании. Очевидно, что метаданные сопутствуют дисциплине «управление метаданными данных».

Многоязычное программирование

Нил Форд (Neal Ford) использует термин «многоязычное программирование» (<https://oreil.ly/65j2B>), чтобы выразить возможность использования в приложениях сочетания языков программирования, потому что разные языки подходят для решения разных задач и у каждого обычно есть свои преимущества.

Модели справочных данных

Некоторые хранилища данных построены на основе готовых стандартных отраслевых моделей данных. IBM (<https://oreil.ly/O8KhU>), Teradata (<https://oreil.ly/sFDdK>), Oracle (<https://oreil.ly/MrQ6K>) и Microsoft (<https://oreil.ly/ZOMfE>), например, предоставляют эти модели данных для различных отраслей.

Моделирование размерности данных

При моделировании или денормализации размерностей данных эти данные сворачиваются, объединяются или группируются. В моделировании размерностей данных используются понятия фактов (мер) и измерений (контекста). Если измерения сворачиваются в отдельные структуры, модель данных также часто называют *звездообразной схемой*. Если измерения не сворачиваются, модель данных называется *снежинкой*. Моделирование размерностей часто используется в системах хранилищ данных.

Моделирование хранилища данных

Моделирование хранилища данных — это метод моделирования БД для разработки и обеспечения долгосрочного хранения данных, поступающих из нескольких операционных систем.

Модель данных

Абстрактная модель, описывающая способы представления и использования данных.

Мультисхема

Некоторые из продуктов баз данных NewSQL также сочетают в себе несколько характеристик схемы NoSQL и предоставляют несколько способов хранения данных и управления ими. Они называются *мультисхемными* или *многомодельными* БД. Пример: Mark-Logic (<https://www.marklogic.com/>), который объединяет множество различных шаблонов проектирования баз данных в единую БД. Например, у вас могут быть одновременно строго согласованные реляционные БД, БД документов, пары «ключ — значение» и графы. Хотя основная идея кажется элегантной, все это усложняет процесс и снижает производительность.

Представление одних и тех же данных с использованием нескольких моделей, усложняет ситуацию, потому что любое изменение в структуре БД влияет на все представления. Изменение основной структуры может привести к появлению различий в представлении данных разными конечными точками. Поэтому разработчикам приходится прилагать больше сил, чтобы все приложения работали согласованно с различными конечными точками БД.

Другая проблема — производительность: многомодельные БД в основном находятся под капотом нескольких различных оптимизированных баз данных. Чем больше у вас представлений, тем большее количество мест приходится копировать данные, тем больше требуется оптимизаций предварительной обработки и индексов, тем больше нужно хранилища, больше пропускной способности и т. д. Некоторые поставщики решают эту проблему, не копируя данные, а преобразуя их в реальном времени по мере выполнения. Этот подход тоже неплохой, но выполнение сложных операторов SQL при внутреннем преобразовании NoSQL в модель данных SQL требует невероятной производительности, что значительно увеличивает стоимость оборудования.

Нераспределенные реляционные базы данных

Нераспределенные RDBM (системы управления реляционными базами данных) основаны на предопределенной реляционной модели с использованием таблиц со строками и столбцами. Нераспределенные реляционные БД совместно используют все хранилище с одним процессором. Язык запросов по умолчанию для систем управления реляционными БД — SQL. Примеры: MySQLdb, MariaDB PostgreSQL, SQL Server, Oracle и IBM Db2.

Нормализация базы данных

Нормализация базы данных — это процесс проектирования для уменьшения избыточности данных и улучшения их целостности.

Область подготовки

Область подготовки — это промежуточная область хранения, используемая для обработки данных во время процесса извлечения, преобразования и загрузки (ETL).

Озеро данных

Озеро данных обычно представляет собой единое хранилище всех корпоративных данных, включая необработанные копии данных из систем-источников и преобразованные данные, используемые для таких задач, как отчетность, визуализация, продвинутая аналитика и машинное обучение.

Операционные метаданные

Операционные метаданные, или метаданные процесса, содержат информацию о выполнении процесса, регистрации и аудита. Примерами могут быть

потоки процессов, отметки времени транзакции и происхождение данных — другими словами, как данные перемещаются между системами.

Оптимизация запросов

Оптимизация запросов определяет наиболее эффективный способ выполнения выбранного запроса, учитывая возможные запросы и планы. Индексы и материализованные представления — тоже часть этого подхода.

Основные (мастер-) данные

Наиболее важные данные называются *мастер-данными*, а сопутствующая дисциплина *управления основными данными* обеспечивает доступность, безопасность, прозрачность и надежность основных данных внутри организации.

Отображение данных

Отображение данных — это процесс создания сопоставлений элементов данных между двумя различными моделями. Это действие считается частью интеграции данных.

Поставщик идентификации

Поставщик идентификации (identity provider, IDP), или поставщик удостоверений, управляет БД удостоверений (пользовательские записи, учетные данные) и предоставляет службу проверки подлинности другим приложениям. За последнее десятилетие такие решения, как OpenLDAP и Microsoft Active Directory, служили основными поставщиками удостоверений для многих организаций. Появились такие протоколы нового поколения, как OpenID Connect и OAuth 2.0, которые упрощают интеграцию компаний.

Протокол сериализации структурированных данных

Протоколы сериализации структурированных данных (Protobufs) (<https://oreil.ly/f0ZKJ>) были разработаны Google как альтернатива XML. Они были оптимизированы для протокола запрос/ответ с целью уменьшения задержки. Протокол сериализации также использует язык описания интерфейса (interface description language, IDL), сравнимый с Apache Thrift. gRPC (<https://grpc.io/>), фреймворк удаленного вызова процедур (RPC), также разработанный Google, использует протокол NoSQL, буферизует и обеспечивает аутентификацию, обработку тайм-аутов, управление потоком и многое другое.

Разграничение доступа на основе атрибутов

Управление доступом на основе атрибутов (ABAC) — это более продвинутый метод безопасности, в котором используются политики, основанные на комбинации различных атрибутов (данных).

Распределенные реляционные базы данных

Распределенные базы данных – это БД, в которых не все устройства хранения подключены к общему процессору и контролируются системой управления распределенной БД. Распределенный кластер узлов без общего доступа использует реляционную модель данных, которая может обрабатывать SQL. Примерами служат Amazon Redshift, Google Big Table (BigQuery), Azure Synapse Analytics, Snowflake, Exadata, Teradata, Greenplum и SAP HANA.

Расширенный протокол организации очереди сообщений

Расширенный протокол организации очереди сообщений (Advanced Message Queuing Protocol, AMQP) – это двоичный протокол, предназначенный для поддержки шаблонов связи и широкого спектра приложений обмена сообщениями. Он совместим со многими платформами вроде Apache ActiveMQ, Azure Event Hubs и RabbitMQ. Kafka не поддерживает AMQP напрямую, хотя есть некоторые доступные компоненты с открытым исходным кодом, которые могут служить мостом (<https://oreil.ly/YZcOv>) для взаимодействия.

Резервная копия данных

Резервная копия данных – это моментальная копия системы (данных и кода), которая обеспечивает ее постоянную доступность в случае аппаратного или программного сбоя. Резервные копии данных используются для восстановления и могут быть созданы для каждого этапа данных.

Реляционное моделирование

Реляционное моделирование – это популярный метод, позволяющий уменьшить дублирование данных и обеспечить их ссылочную целостность. ЗНФ, например, гарантирует, что каждая сущность не имеет скрытых первичных ключей и что каждый атрибут не зависит от атрибутов вне ключа. Эта модель обычно используется в операционных и транзакционных системах.

Репозиторий метаданных

Репозиторий метаданных – это программный инструмент или БД, используемые для хранения метаданных и управления ими.

Самостоятельная бизнес-аналитика

Самостоятельная бизнес-аналитика позволяет бизнес-пользователям самостоятельно получать доступ к данным и работать с ними вместо передачи требований в ИТ-отдел для выполнения.

Своевременность

Своевременность – это степень, в которой данные представляют временную реальность. Своевременность можно использовать, чтобы оценить доступность данных и определить, были ли данные доставлены слишком поздно.

Сегментирование

Сегментирование (sharding) — это процесс разделения данных на отдельные хранилища данных или горизонтальные разделы на основе некоторого логического признака, такого как регион, сегменты клиентов и т. д. Этот шаблон может улучшить масштабируемость при хранении и доступе к большим объемам данных.

Секционирование

Секционирование — это распределение данных по нескольким файлам в кластере для балансировки больших объемов данных по дискам или узлам. Разделы только для чтения составляют табличное пространство, доступное только для чтения. Это предотвращает обновления всех таблиц в табличном пространстве. К этому пространству можно применить другие шаблоны для повышения производительности.

Система записей

Термин «системы записей» часто используется для обозначения исходных данных¹. Системы записей применяются для классификации систем или БД, которые являются авторитетными источниками данных для определенных данных или информации.

Системы управления базами данных массивов

Системы управления БД массивов рассчитаны на работу со структурами данных в форме массивов. Структура данных в форме массива, или просто *массив*, хранит данные в структуре, напоминающей растр или сетку. БД массивов обычно используются для сложной аналитики или обнаружения данных в больших и сложных коллекциях данных. Примеры: SciDB, MonetDB/SciQL и PostGIS.

Скремблирование данных

Скремблирование данных — это процесс запутывания или удаления конфиденциальных данных. Он необратим, поэтому исходную конфиденциальную информацию невозможно извлечь из скремблированных данных. Скремблирование данных можно использовать только во время процесса дублирования данных.

Совместимость данных

Совместимость данных — это способность нескольких приложений взаимодействовать (и обмениваться данными).

Согласованность

Степень, в которой данные соответствуют определению данных. Примером может служить поле имени человека, которое представляет имя, фамилию или комбинацию имени и фамилии.

¹ Inmon B. The System of Record in the Global Data Warehouse.

Согласованность базы данных

Члены сообщества баз данных по-разному определяют слово «согласованность». *Слабо согласованное* или *согласованное в конечном итоге* чтение означает, что для строк таблицы не установлены блокировки и поэтому другие процессы могут изменять данные, пока вы читаете их. Следовательно, полученные вами данные могут не отражать фактические результаты недавно завершенной операции записи. Когда чтение *строго согласовано*, данные блокируются и процессам приходится ждать, поэтому всегда возвращаются самые свежие данные.

Согласованность записи также имеет разные значения. Для распределенных систем *строгая согласованность* гарантирует успех операции записи, а также то, что все (или как минимум два) соответствующие узлы в распределенной системе будут обновлены правильно. *Согласованные в конечном итоге* операции записи работают противоположным образом. Они не гарантируют успешного обновления данных на всех узлах, если система выйдет из строя.

Соль

Соль — это процесс рандомизации данных, чтобы их можно было использовать в качестве дополнительных входных данных для односторонней функции, которая хеширует данные, пароли или парольные фразы.

Специализированное оборудование

Специализированным называется выделенное или специальное оборудование, ориентированное на улучшение обслуживания определенных рабочих нагрузок. Например, известно, что графические процессоры (GPU) обрабатывают сложные структуры данных параллельно быстрее, чем традиционные центральные процессоры (CPU).

Справочная информация

Справочная информация обычно используется для связи и предоставления дополнительных сведений к данным. Это данные, применяемые для классификации, организации или категоризации других данных. Справочные данные могут содержать иерархии значений, например отношения между иерархиями продуктов и географическими иерархиями. Существует дисциплина *управление справочными данными*, которая обеспечивает согласованность справочных данных, а также правильное управление и распределение различных версий.

Строгая согласованность

Строгая согласованность гарантирует, что данные будут записаны ровно один раз. Сильная согласованность не может обеспечить 100%-ную доступность, поэтому может вернуть ошибку, если что-то пойдет не так, что потребует от клиента повторной отправки данных позже. Выбор согласованности означает, что вы должны согласиться с тем, что возвращается наиболее несовместимое

значение или что во время сбоя системы данные могут стать несогласованными.

Строительный блок решения

Строительный блок решения (solution building block, SBB) – это группа некоторых функций в контексте реализации с четко определенными функциями и отношениями, например реализация системы CRM в контексте розничного отдела.

Структурирование данных при записи в БД

При использовании схемы во время записи (целевая) схема известна и должна быть создана до того, как какие-либо данные будут записаны в БД. Структурирование при чтении отличается от структурирования при записи, поскольку схема базы данных создается в момент чтения данных и еще не должна существовать.

Текущие данные

Текущие данные – это самая последняя и актуальная версия данных. Приложение часто обращается к ним. Обновления текущих данных ожидаются регулярно. Они напрямую используются приложениями и бизнес-процессами.

Технические метаданные

Технические метаданные касаются технических систем, интерфейсов или баз данных. Примерами являются схемы баз данных и атрибуты интерфейсов.

Токенизация

Токенизация – это процесс замены конфиденциальных данных уникальными идентификационными символами (токенами), которые сохраняют всю важную информацию о данных, не угрожая их безопасности.

Токены доступа

Токен доступ используется для представления личности пользователя или группы пользователей вместо имени и пароля. Он может содержать дополнительные атрибуты и абстракции, описывающие контекст использования токена или интервал времени, в течение которого токен остается действительным.

Точность

Степень, в которой данные отражают правду или реальность. Орфографическая ошибка – хороший пример неточных данных.

Трансформация данных

Трансформация данных – это процесс согласования выбранных данных со структурой целевой БД. Примеры: изменения формата, изменения струк-

туры, семантические или контекстные изменения, удаление дубликатов и переупорядочение.

Трехуровневая архитектура

Трехуровневая архитектура — это шаблон архитектуры программного обеспечения «клиент — сервер», в котором пользовательский интерфейс (представление), функциональная логика процесса (бизнес-правила), компьютерное хранилище данных и доступ к данным разрабатываются и поддерживаются как независимые модули, чаще всего как отдельные площадки.

Уникальность

Степень уникальности данных (хранятся в единственном экземпляре). Хорошие примеры — повторяющиеся записи или одинаковые ключи.

Управление доступом на основе ролей

Управление доступом на основе ролей (role-based access control, RBAC) — это метод безопасного доступа, основанный на определении ролей и соответствующих привилегий.

Уровень бизнес-логики

Уровень бизнес-логики передает информацию (данные) между уровнем представления и уровнем данных. Это взаимодействие может включать сложную проверку и бизнес-логику (правила принятия решений, преобразования, вычисления, агрегации и сложные статистические модели). Этот уровень может быть разработан с использованием множества различных методов и языков программирования. Разработчики и инженеры часто выбирают технологии, исходя из своих предпочтений.

Уровень данных

Уровень данных (хранилище), также известный как хранилище данных, отвечает за сохранение (хранение) данных приложения. Обычно используются базы данных, но простой файл, содержащий данные приложения, тоже может выполнять эту работу. Тип ядра БД зависит от структур данных, используемых приложением, гибкости и нефункциональных требований, таких как производительность.

Уровень представления

Уровень представления или уровень пользовательского интерфейса отвечает за взаимодействие и операции пользователя. Обычно он представляет результаты работы приложения и выполняет взаимодействие в удобной для пользователя форме. Современные приложения часто взаимодействуют через веб-браузер или мобильные приложения. Основными технологиями являются веб-технологии, такие как HTML5, JavaScript и каскадные таблицы стилей (CSS).

Функция приложения

Составная часть модели архитектуры, описывающая единую, не зависящую от продукта и многократно используемую функцию всей модели. Например, функция аутентификации, которая может быть основана на отраслевых стандартах, таких как SAML и OpenID Connect.

Хранилища операционных данных

Хранилища операционных данных (operational data store, ODS) обычно используются для оперативного анализа и отчетности. ODS обладают некоторыми характеристиками систем OLTP и OLAP. Слово «*операционные*» приближает ODS к OLTP-системам, потому что их цель — получить представление об эксплуатационных характеристиках и деятельности. В основном они наследуют контекст от OLTP-систем.

Характеристики, унаследованные от систем OLAP, следующие: ODS устраняют аналитические рабочие нагрузки, которые могут быть вызваны специальным, менее предсказуемым анализом и отчетностью. Другая характеристика заключается в том, что ODS в целом хранят больше исторических данных, чем системы OLTP. ODS также могут интегрировать небольшую часть данных из других систем. Это означает, что при разработке ОРВ также может быть задействован аспект интеграции или гармонизации.

В целом ODS остаются ближе к конструкции первичной системы OLTP, с которой они могут работать. Структуры таблиц часто похожи. В результате ODS отличаются от хранилищ данных, поскольку они обычно используют данные только из одной системы OLTP, в отличие от хранилищ данных, в которых хранятся данные из нескольких источников.

Хранилище данных

Хранилище данных¹ — это предметно-ориентированный, интегрированный, изменяющийся во времени и энергонезависимый набор данных, поддерживающий процесс принятия решений руководством.

Хеширование

Хеширование — это процесс сопоставления значений данных с хеш-значениями фиксированного размера (хешами). Распространенными алгоритмами хеширования являются дайджест сообщений (Message Digest 5, MD5) и алгоритм безопасного хеширования (Secure Hashing Algorithm, SHA). Хеш-значение невозможно преобразовать в исходное значение данных.

¹ Inmon B. Building the Data Warehouse. — Wiley, 1992.

Целостность данных

Степень, в которой данные сохраняют внутреннюю или ссылочную целостность. Если ключ для ссылки на другую таблицу недействителен, соединение между двумя таблицами не может быть выполнено.

Шифрование

Шифрование — это перевод данных в сложные коды, которые нельзя интерпретировать (расшифровать) без использования ключа дешифрования. Обычно эти ключи распространяются и хранятся отдельно. Существует два типа шифрования: *шифрование с симметричным ключом* и *шифрование с открытым ключом*. При шифровании с симметричным ключом ключи шифрования и дешифрования абсолютно одинаковые. Шифрование с открытым ключом имеет два разных ключа. Один ключ используется для шифрования значений (открытый ключ), а другой — для расшифровывания данных (закрытый ключ).

Эластичность

Эластичность — это степень, в которой системы могут адаптировать изменения своей рабочей нагрузки за счет автоматического выделения и отключения ресурсов.

Эталонные модели

Эталонные модели, вроде моделей приложений и возможностей, помогают ЕА упростить, классифицировать и определить, как возможности приложений поддерживают конкретные бизнес-цели. Приложения и бизнес-функции могут быть визуализированы как структура, показывающая, какие независимые функции подлежат повторному использованию.

Язык описания данных

Физическая модель данных приложения (application physical data model, APDM), описанная в проекте, может отличаться от фактической реализации. Для создания и изменения структуры объектов БД в базе данных обычно требуется использовать язык определения данных (data definition language, DDL). Многие инструменты моделирования данных создают описания на языке DDL, но имейте в виду, что APDM и DDL — это не одно и то же.

Операторы DDL также могут использоваться для принудительного шифрования данных (маскировки или обfuscации), что делает невозможным чтение базы данных другими пользователями. Следовательно, физическая реализация и проект физического представления могут различаться.

06 авторе

Питхейн Стрендхолт (Piethein Strengtholt) — главный архитектор ABN AMRO — курирует стратегию обработки данных и изучает ее влияние на деятельность организации. Ранее он работал стратегическим консультантом, проектировал множество архитектур и участвовал в крупных программах управления данными, а также был внештатным разработчиком приложений. Живет в Нидерландах со своей семьей.

Об обложке

На обложке изображена зеленая европейская ящерица (*Lacerta viridis*), которую можно встретить на юго-востоке одноименного континента. Эти ящерицы живут в густых насаждениях и кустарниках, где могут легко выходить на солнце и ловить насекомых и мелких беспозвоночных. Иногда они едят фрукты, птичьи яйца, более мелких сородичей и мышей.

На горле у европейских зеленых ящериц расположены голубоватые чешуйки, особенно ярко они выражены у самцов. Ящерицы могут вырасти до 40 сантиметров, причем хвост составляет две трети длины. В качестве защиты ящерицы могут отбрасывать хвост. После этого в течение нескольких недель у них вырастает новый. Эти ящерицы классифицируются как вызывающие наименьшее опасение с точки зрения сохранения вида.

Многие животные на обложках O'Reilly находятся под угрозой исчезновения, все они важны для мира. Чтобы узнать о том, как вы можете помочь, перейдите на сайт animals.oreilly.com.

Иллюстрация на обложке выполнена Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры из *Encyclopedie D'Histoire Naturelle*.

Питхейн Стренхольм

**Масштабируемые данные.
Лучшие шаблоны высоконагруженных архитектур**

Перевел с английского С. Черников

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Н. Гринчик</i>
Научный редактор	<i>А. Киселев</i>
Литературный редактор	<i>Т. Сажина</i>
Художественный редактор	<i>В. Мостиан</i>
Корректоры	<i>Е. Павлович, Н. Терех</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес:
194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 07.2022. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —

Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 27.05.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 29,670. Тираж 700. Заказ 0000.