O'REILLY®

Распределенные данные

Алгоритмы работы современных систем хранения информации

Алекс Петров

Алекс Петров

Распределенные данные. Алгоритмы работы современных систем хранения информации

Серия «Бестселлеры O'Reilly»
Перевел на русский А. Коцюба

Руководитель проекта	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Научный редактор	*А. Коцюба*
Литературный редактор	*А. Руденко*
Обложка	*В. Мостипан*
Корректоры	*Н. Петрова, М. Одинокова*
Верстка	*Е. Неволайнен*

ББК 32.973.233-018
УДК 004.65

Петров Алекс

П30 Распределенные данные. Алгоритмы работы современных систем хранения информации. — СПб.: Питер, 2021. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1640-9

Когда дело доходит до выбора, использования и обслуживания базы данных, важно понимать ее внутреннее устройство. Как разобраться в огромном море доступных сегодня распределенных баз данных и инструментов? На что они способны? Чем различаются? Алекс Петров знакомит нас с концепциями, лежащими в основе внутренних механизмов современных баз данных и хранилищ. Для этого ему пришлось обобщить и систематизировать разрозненную информацию из многочисленных книг, статей, постов и даже из нескольких баз данных с открытым исходным кодом. Вы узнаете об принципах и концепциях, используемых во всех типах СУБД, с акцентом на подсистеме хранения данных и компонентах, отвечающих за распределение. Эти алгоритмы используются в базах данных, очередях сообщений, планировщиках и в другом важном инфраструктурном программном обеспечении. Вы разберетесь, как работают современные системы хранения информации, и это поможет взвешенно выбирать необходимое программное обеспечение и выявлять потенциальные проблемы.

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

ISBN 978-1492040347 англ. Authorized Russian translation of the English edition of Database Internals
ISBN 9781492040347 © 2020 Alex Petrov
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.

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: 07.2021.
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 02.07.21. Формат 70x100/16. Бумага офсетная. Усл. п. л. 27,090. Тираж 500. Заказ

Оглавление

ЧАСТЬ I

Подсистема хранения данных

Основной задачей любой СУБД является надежное хранение данных и обеспечение доступа к ним со стороны пользователей. Мы используем базы данных в качестве основного источника данных, помогающего обмениваться данными между различными частями приложений. Вместо того чтобы искать способ хранения и извлечения информации и изобретать новый способ организации данных при создании каждого нового приложения, мы используем базы данных. Таким образом, мы можем сосредоточиться на логике приложений, а не на инфраструктуре.

Для большей компактности вместо громоздкого термина «система управления базами данных» на протяжении этой книги мы будем в основном употреблять соответствующую аббревиатуру (СУБД) или просто выражение «база данных», имея в виду то же самое.

База данных представляет собой модульную систему, включающую в себя несколько составных частей: транспортный уровень, который принимает запросы, обработчик запросов, который выбирает наиболее эффективный способ выполнения запросов, подсистему выполнения, которая производит выполнение операций, и подсистему хранения (см. раздел «Архитектура СУБД» на с. 25).

Подсистема хранения данных (или ядро СУБД) — это программный компонент СУБД, отвечающий за хранение, извлечение и управление данными в памяти и на диске, предназначенный для работы с постоянной, долговременной памятью каждого узла [REED78]. В то время как базы данных могут отвечать на сложные запросы, подсистемы хранения рассматривают данные более детально и предлагают простой API для манипуляций с данными, позволяющий пользователям создавать, обновлять, удалять и извлекать записи. СУБД можно рассматривать как приложение, которое надстроено поверх подсистемы хранения и предлагает некоторую схему данных, язык запросов, индексацию, транзакции и много других полезных функций.

Для обеспечения гибкости и ключи, и значения могут быть произвольными последовательностями байтов без заданной формы. Их классификация и семантика представления определяются в подсистемах более высокого уровня. Например, вы можете использовать `int32` (32-разрядное целое число) в качестве ключа в одной из

таблиц и `ascii` (строку в кодировке ASCII) — в другой; с точки зрения подсистемы хранения оба ключа являются просто сериализованными записями.

Такие подсистемы хранения, как BerkeleyDB (https://databass.dev/links/92), LevelDB (https://databass.dev/links/93) и ее потомок RocksDB (https://databass.dev/links/94), LMDB (https://databass.dev/links/95) и ее потомок libmdbx (https://databass.dev/links/96), Sophia (https://databass.dev/links/97), HaloDB (https://databass.dev/links/98) и многие другие, были разработаны независимо от тех СУБД, в которые они теперь встроены. Использование существующих подключаемых подсистем хранения позволило разработчикам СУБД обойти этап их создания и сосредоточиться на других подсистемах.

В то же время четкое разделение между компонентами СУБД открывает возможность переключаться между различными подсистемами, потенциально более подходящими для конкретных сценариев использования. Например, MySQL, популярная система управления базами данных, имеет несколько подсистем хранения (https://databass.dev/links/99), включая InnoDB, MyISAM и RocksDB (https://databass.dev/links/100) (в составе MyRocks (https://databass.dev/links/101)). MongoDB позволяет переключаться между подсистемами хранения WiredTiger (https://databass.dev/links/102), In-Memory и (теперь уже устаревшей) MMAPv1 (https://databass.dev/links/103).

Сравнение баз данных

Выбор той или иной СУБД может иметь долгосрочные последствия. Если есть вероятность того, что база данных не подойдет из-за проблем с производительностью, обеспечением согласованности или выполнением операций, лучше выяснить это на раннем этапе цикла разработки, так как миграция на другую систему может оказаться нетривиальной. В некоторых случаях для этого потребуются существенные изменения в коде приложения.

У каждой СУБД есть свои сильные и слабые стороны. Для снижения риска дорогостоящей миграции можно потратить некоторое время перед принятием решения о выборе конкретной базы данных, чтобы быть уверенным в ее соответствии потребностям вашего приложения.

Попытка сравнить базы данных исходя из их компонентов (например, какую подсистему хранения они используют, каким образом данные совместно используются, реплицируются и распределяются и т. д.), их рейтинга (субъективная оценка, сделанная консалтинговыми агентствами, такими как ThoughtWorks (https://www.thoughtworks.com/de/radar), или сайтами со сравнением баз данных, такими как DB-Engine (https://db-engines.com/de/ranking) или Database of Databases (https://dbdb.io/)) или языка реализации (C++, Java или Go и т. д.) может привести к неверным и поспешным выводам. Такие методы можно использовать только для предварительного сравнения, и они могут быть такими же грубыми, как выбор между HBase и SQLite, поэтому даже поверхностное понимание того, как работает каждая база данных и что находится внутри нее, поможет принять более взвешенное решение.

Каждое сравнение должно начинаться с четкого определения цели, потому что даже малейшая предвзятость может полностью свести на нет все исследование. Если вам нужно, чтобы база данных хорошо подходила для имеющихся у вас (или ожидаемых) рабочих нагрузок, то лучше всего будет смоделировать эти рабочие нагрузки для различных СУБД, измерить важные для вас показатели производительности и сравнить результаты. Некоторые проблемы, особенно когда речь заходит о производительности и масштабируемости, начинают проявляться только через некоторое время или по мере роста емкости. Для выявления потенциальных проблем лучше всего провести длительные тесты в среде, максимально точно имитирующей реальное рабочее окружение.

Моделирование реальных рабочих нагрузок не только помогает понять, как работает база данных, но и позволяет научиться ее использовать и отлаживать, а также выяснить, насколько дружелюбно и полезно сложившееся вокруг нее сообщество. Выбор базы данных всегда определяется сочетанием этих факторов, и производительность часто оказывается не самым важным аспектом: лучше использовать базу данных, которая медленно сохраняет данные, а не быстро их теряет.

Чтобы сравнить базы данных, полезно тщательно изучить сценарий использования и определить текущие и ожидаемые переменные:

- размеры схемы данных и записей;
- количество клиентов;
- типы запросов и паттерны доступа;
- скорость выполнения запросов на чтение и запись;
- ожидаемые изменения в любой из этих переменных.

Знание этих переменных может помочь ответить на следующие вопросы:

- Поддерживает ли база данных необходимые запросы?
- Способна ли база данных обрабатывать тот объем данных, который мы планируем хранить?
- Сколько операций чтения и записи может обрабатывать один узел?
- Сколько узлов должна включать в себя система?
- Как расширить кластер с учетом ожидаемых темпов роста?
- В чем заключается процесс обслуживания?

Получив ответы на эти вопросы, вы сможете построить тестовый кластер и смоделировать свои рабочие нагрузки. Для большинства баз данных уже есть инструменты стресс-анализа, которые можно применить для воспроизведения конкретных сценариев использования. Если в экосистеме базы данных нет стандартного инструментария стресс-анализа для создания реалистичных рандомизированных рабочих нагрузок, это может стать настораживающим знаком. Если что-то мешает вам использовать инструменты, поставляемые вместе с самой базой, можно попро-

бовать один из существующих инструментов общего назначения или реализовать его с нуля.

Если тесты показывают положительные результаты, часто полезно ознакомиться с кодом базы данных. При изучении кода обычно лучше сначала выявить составные части базы данных, понять, как найти код различных компонентов, а затем пройтись по ним. Имея даже приблизительное представление о кодовой базе СУБД, вы сможете лучше разбираться в создаваемых ею записях журнала и ее параметрах конфигурации, а также выявлять проблемы в использующем ее приложении и даже в самом коде СУБД.

Было бы здорово, если бы мы могли использовать базы данных как черные ящики и никогда не заглядывать в них, но практика показывает, что рано или поздно возникает ошибка, сбой, регрессия производительности или какая-то другая проблема и лучше быть готовым к этому. Если вы знаете и понимаете внутренние компоненты базы данных, то можете снизить бизнес-риски и повысить шансы на быстрое восстановление.

Одним из популярных инструментов для тестирования, оценки производительности и сравнения является Yahoo! Cloud Serving Benchmark (YCSB) (https://databass.dev/links/104). YCSB предлагает инфраструктуру и общий набор рабочих нагрузок, которые могут быть применены к различным хранилищам данных. Как и все средства общего назначения, этот инструмент следует использовать с осторожностью, так как можно с легкостью прийти к неправильным выводам. Чтобы провести справедливое сравнение и принять обоснованное решение, необходимо потратить достаточно времени, чтобы понять реальные условия, в которых должна функционировать база данных, и соответствующим образом адаптировать сравнительные тесты.

СРАВНИТЕЛЬНЫЙ ТЕСТ TPC-C

Совет по оценке производительности обработки транзакций (Transaction Processing Performance Council, TPC) предлагает набор сравнительных тестов, которые поставщики баз данных используют для сравнения и рекламирования производительности своих продуктов. TPC-C — это тест обработки транзакций в реальном времени (Online Transaction Processing, OLTP), представляющий собой смесь транзакций только для чтения и обновления, которые имитируют распространенные рабочие нагрузки приложений.

TPC-C тестирует производительность и корректность выполняемых конкурентных транзакций. Основным показателем производительности является пропускная способность: количество транзакций, которые СУБД может обрабатывать в минуту. Выполняемые транзакции должны сохранять ACID-свойства и соответствовать (не противоречить) набору свойств, определяемых самим тестом.

Этот тест не относится к какому-либо конкретному бизнес-сегменту, но предоставляет абстрактный набор действий, важных для большинства приложений, для которых подходят базы данных OLTP. Он включает в себя несколько таблиц и объектов, таких как склады, товарные запасы, заказчики и заказы, и определяет макеты таблиц, сведения о транзакциях, которые могут быть выполнены с этими таблицами, минимальное количество строк в таблице и ограничения на уровень устойчивости данных.

Это не означает, что сравнительные тесты можно использовать *только* для сравнения баз данных. Сравнительные тесты могут быть полезны для определения и проверки положений соглашения об уровне обслуживания[1], определения системных требований, планирования производительности и многого другого. Чем больше вы узнаете о базе данных до ее использования, тем меньше времени придется тратить при эксплуатации.

Выбор базы данных — это долгосрочное решение, и потому желательно отслеживать выход новых версий, понимать, что именно изменилось и почему, и иметь некоторую стратегию обновления. Новые выпуски обычно содержат улучшения и исправления ошибок и проблем безопасности, но могут содержать и новые ошибки, вести к снижению производительности или неожиданному поведению, поэтому новые версии тоже важно тестировать перед их развертыванием. Проверка того, как разработчики базы данных проводили обновления ранее, может дать вам хорошее представление о том, чего ожидать в будущем. Если предыдущие обновления проходили гладко, это еще не гарантирует, что также будет и в дальнейшем, однако если они вызывали затруднения, это может быть признаком того, что будущие обновления тоже дадутся нелегко.

Понимание преимуществ и недостатков

В качестве пользователей мы можем видеть, как базы данных ведут себя в различных условиях, но при разработке баз данных мы должны принимать решения, оказывающие непосредственное влияние на это поведение.

Проектирование подсистемы хранения, безусловно, сложнее, чем просто реализация структуры данных из учебника: здесь имеется много деталей и пограничных случаев, с которыми обычно не удается справиться с первого раза. Нужно спроектировать физический макет данных и организовать указатели, принять решение о формате сериализации, понять, как данные будут удаляться при сборке мусора, как подсистема хранения вписывается в семантику СУБД в целом, выяснить, как заставить ее работать в конкурентном окружении, и, наконец, убедиться, что мы не будем терять данные ни при каких обстоятельствах.

Помимо того что вам нужно принять решения в отношении множества различных вещей, ситуация осложняется еще и тем, что в большинстве случаев эти решения подразумевают нахождение некоторого компромисса. Например, если мы будем сохранять записи в том же порядке, в котором они вносятся в базу данных, их можно будет сохранять быстрее, но если при этом их нужно будет извлекать в лексикографическом порядке, то их потребуется пересортировывать перед возвращением

[1] Соглашение об уровне обслуживания (service-level agreement, SLA) — это обязательство поставщика услуг относительно качества предоставляемых услуг. Помимо прочего, такое соглашение может включать информацию о задержке, пропускной способности, дрожании, а также количестве и частоте отказов.

результатов клиенту. Как вы увидите на протяжении этой книги, существует много различных подходов к разработке подсистемы хранения и каждая реализация имеет свои собственные плюсы и минусы.

При рассмотрении различных подсистем хранения мы будем обсуждать все их преимущества и недостатки. Если бы существовала подсистема хранения, идеальным образом подходящая для каждого возможного сценария использования, то все бы просто использовали его. Но поскольку такой подсистемы хранения не существует, мы должны подходить к выбору очень тщательно, принимая в расчет ожидаемые рабочие нагрузки и сценарии использования.

Существует множество различных подсистем хранения, использующих разные структуры данных и реализованных на разных языках, начиная с таких низкоуровневых языков, как C, и заканчивая такими высокоуровневыми языками, как Java. В то же время все подсистемы хранения сталкиваются с одинаковыми проблемами и ограничениями. Это можно сравнить с планированием городской застройки — планируя застройку под конкретное количество людей, вы можете увеличивать город в вертикальном или горизонтальном направлении. Хотя в обоих случаях город сможет принять одинаковое количество людей, эти подходы подразумевают совершенно разный образ жизни. При вертикальном росте застройки люди будут жить в квартирах и высокая плотность населения, вероятно, приведет к высокой интенсивности транспортного движения. С другой стороны, при менее плотном размещении люди, вероятно, будут жить в домах, но дальше от места работы.

Аналогичным образом проектные решения, принимаемые разработчиками подсистем хранения, делают их более подходящими для различных целей: одни из них обеспечивают минимальное время чтения и записи, другие — максимальную плотность (количество хранимых данных, приходящееся на каждый узел), а третьи — максимальную простоту эксплуатации.

В разделе «Итоги» в конце каждой главы вы найдете ссылки на полное описание применяемых в реализации алгоритмов и другую полезную информацию. Чтение этой книги должно хорошо подготовить вас для эффективного использования этих источников и дать прочное понимание того, какие имеются альтернативы для описываемых в них концепций.

ГЛАВА 1

Введение и обзор

СУБД служат различным целям: некоторые используются в основном для временных «горячих» данных, некоторые служат в качестве долговременного хранилища «холодных» данных, некоторые подходят для сложных аналитических запросов, некоторые позволяют получать доступ к значениям только по ключу, некоторые оптимизированы для хранения данных временных рядов, а некоторые эффективно хранят большие двоичные объекты. Сначала мы рассмотрим и обозначим различия, начав с краткой классификации и обзора, поскольку это поможет оценить масштаб дальнейшей дискуссии.

Терминология иногда может быть неоднозначной и трудной для понимания без принятия во внимание полного контекста. Примером могут служить различия между *колоночными* хранилищами и хранилищами *с широкими столбцами*, которые имеют очень мало общего между собой, или взаимосвязь *кластеризованных* или *некластеризованных индексов* и *индексированных таблиц.* Цель данной главы состоит в том, чтобы дать однозначное и точное определение таких терминов.

Мы начнем с обзора архитектуры СУБД (см. раздел «Архитектура СУБД» на с. 25) и обсудим отдельные ее компоненты и их функции. После этого мы осветим различия между различными СУБД с точки зрения используемой среды хранения (см. раздел «Резидентные и дисковые СУБД» на с. 27) и структуры (см. раздел «Колоночные и строчные СУБД» на с. 30).

Эти два способа разделения являются далеко не единственными вариантами классификации СУБД. Например, СУБД часто разделяют на три основные категории:

Базы данных для обработки транзакций в реальном времени (*online transaction processing, OLTP*)

Они обрабатывают большое количество поступающих со стороны пользователя запросов и транзакций. Запросы часто бывают предопределенными и с коротким жизненным циклом.

Базы данных для аналитической обработки данных в реальном времени (*online analytical processing, OLAP*)

Они обрабатывают сложные агрегаты данных. OLAP-системы часто используются для аналитики и построения хранилищ данных и способны обрабатывать сложные произвольные специальные запросы с длительным жизненным циклом.

Гибридная транзакционно-аналитическая обработка (*hybrid transactional and analytical processing, HTAP*)

Эти базы данных сочетают в себе свойства хранилищ OLTP и OLAP.

Существует и множество других терминов и способов классификации: хранилища типа «ключ–значение», реляционные базы данных, документоориентированные хранилища и графовые базы данных. Эти понятия здесь не определяются, так как предполагается, что читатель хорошо знает и понимает их. Поскольку концепции, которые мы здесь обсуждаем, широко применимы и используются в большинстве упомянутых типов хранилищ в том или ином качестве, полная систематика не является необходимой или важной для дальнейшего обсуждения.

Часть I этой книги посвящена структурам хранения и индексирования, поэтому нам необходимо понять высокоуровневые подходы к организации данных и взаимосвязь файлов данных и индексных файлов (см. раздел «Файлы данных и индексные файлы», с. 34).

Наконец, в разделе «Буферизация, неизменяемость и упорядочение» на с. 39 мы обсудим три широко используемых метода разработки эффективных структур хранения данных, а также поговорим о том, как применение этих методов влияет на проектирование и реализацию структур.

Архитектура СУБД

Не существует общего шаблона для проектирования СУБД. Каждая база данных строится немного по-разному, а границы компонентов довольно трудно обнаружить и определить. Даже если эти границы существуют на бумаге (например, в проектной документации), в коде кажущиеся независимыми компоненты могут оказаться связанными из-за оптимизации производительности, обработки граничных случаев или применения определенных архитектурных решений.

Источники, описывающие архитектуру СУБД (например, [HELLERSTEIN07] [WEIKUM01] [ELMASRI11] и [MOLINA08]), определяют компоненты и отношения между ними поразному. Архитектура, показанная на рис. 1.1, демонстрирует некоторые общие элементы этих представлений.

СУБД используют *модель «клиент — сервер»*, в которой экземпляры СУБД (*узлы*) играют роль серверов, а экземпляры приложений — роль клиентов.

Запросы клиентов поступают через *транспортную подсистему*. Поступающие запросы чаще всего оформлены на каком-либо языке запросов. Транспортная подсистема также отвечает за связь с другими узлами кластера баз данных.

После получения запроса транспортная подсистема передает его обработчику запросов, который анализирует, интерпретирует и проверяет его. Далее проводятся проверки управления доступом, так как для их полного выполнения необходима интерпретация запроса.

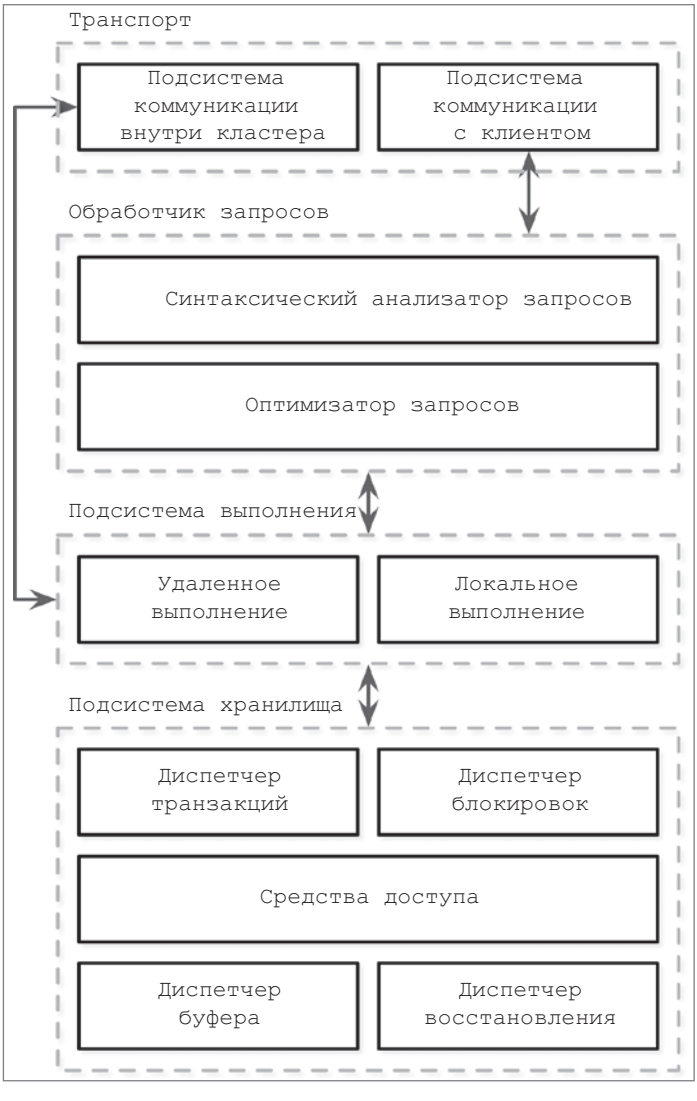

Рис. 1.1. Архитектура СУБД

Анализируемый запрос передается *оптимизатору запросов*, который сначала устраняет невозможные и избыточные части запроса, а затем пытается найти наиболее эффективный способ его выполнения на основе внутренней статистики (мощность индекса, приблизительный размер пересечений и т. д.) и размещения данных (какие узлы в кластере содержат данные и какие затраты требуются для их передачи). Оптимизатор обрабатывает реляционные операции, необходимые для разрешения запросов, обычно представленные в виде дерева зависимостей, и проводит оптимизации, такие как упорядочение индексов, оценка мощности и выбор средств доступа.

Запрос обычно представлен в виде *плана выполнения* (или *плана запроса*): последовательности операций, которую необходимо выполнить для того, чтобы результаты запроса считались полными. Поскольку один и тот же запрос может быть выполнен с помощью различных планов выполнения, которые могут отличаться по эффективности, оптимизатор выбирает наиболее эффективный план из всех доступных.

План выполнения обрабатывается *подсистемой выполнения*, которая собирает результаты выполнения локальных и удаленных операций. *Удаленное выполнение* обычно сводится к записи и чтению данных на других узлах кластера, а также выполнению репликации.

Локальные запросы (поступающие непосредственно от клиентов или от других узлов) выполняются *подсистемой хранения данных*, которая включает в себя несколько компонентов с четко заданными функциями:

Диспетчер транзакций

Производит планировку транзакций и гарантирует, что они не оставят базу данных в логически несогласованном состоянии.

Диспетчер блокировок

Блокирует объекты базы данных для выполняемых транзакций, чтобы конкурентные операции не нарушали физическую целостность данных.

Средства доступа (структуры для хранения данных)

Управляют доступом и организацией данных на диске. Средства доступа включают в себя неупорядоченные файлы и такие структуры хранения, как B-деревья (см. раздел «Вездесущие B-деревья», с. 51) или LSM-деревья (см. раздел «LSM-деревья», с. 148).

Диспетчер буферов

Кэширует страницы данных в памяти (см. раздел «Организация буферизации данных», с. 98).

Диспетчер восстановления

Ведет журнал операций и восстанавливает состояние системы в случае сбоя (см. раздел «Восстановление», с. 106).

Вместе диспетчеры транзакций и блокировок отвечают за управление параллелизмом (см. раздел «Управление параллелизмом», с. 111): они гарантируют логическую и физическую целостность данных, обеспечивая при этом максимально эффективное выполнение конкурентных операций.

Резидентные и дисковые СУБД

СУБД хранят данные в оперативной памяти или на диске. *Резидентные СУБД* (или *СУБД в оперативной памяти*) хранят данные *главным образом* в оперативной памяти, а диск используют для восстановления и ведения журнала. *Дисковые СУБД* хранят

бóльшую часть данных на диске и используют память для кэширования содержимого диска или в качестве временного хранилища. Системы обоих типов в той или иной степени используют диск, но резидентные базы данных хранят содержимое почти исключительно в ОЗУ.

Доступ к памяти был и остается на несколько порядков быстрее доступа к диску[1], поэтому есть смысл использовать память в качестве основного хранилища. Такой подход становится все более экономически целесообразным, поскольку цены на память снижаются. Однако цены на оперативную память по-прежнему остаются высокими по сравнению с постоянными устройствами хранения данных, такими как твердотельные накопители и жесткие диски.

Резидентные СУБД отличаются от дисковых не только основной средой хранения данных, но и тем, как они организованы и какие структуры данных и методы оптимизации они используют.

Использование памяти в качестве основного хранилища данных в таких базах данных обусловлено главным образом высокой производительностью, сравнительно низкими затратами на доступ и высокой гранулярностью доступа. С точки зрения программирования работа с оперативной памятью также представляет гораздо меньше сложностей, чем работа с диском. Операционные системы абстрагируют управление памятью и позволяют нам мыслить в терминах выделения и освобождения областей памяти произвольного размера. На диске же мы должны вручную управлять ссылками на местонахождение данных, форматами сериализации, освобожденной памятью и фрагментацией.

Основными ограничивающими факторами роста количества резидентных баз данных являются непостоянство (т. е. недостаточная долговечность) оперативной памяти и высокая стоимость. Поскольку содержимое ОЗУ не является постоянным, программные ошибки, отказы, аппаратные сбои и перебои в подаче электроэнергии могут привести к потере данных. Существуют способы обеспечения долговечности, такие как использование источников бесперебойного питания и оперативной памяти с резервным аккумуляторным питанием, но они требуют дополнительных аппаратных ресурсов и более высокого уровня квалификации обслуживающего персонала. На практике решающую роль часто играет то, что диски проще в обслуживании и гораздо меньше стоят.

Ситуация, вероятно, будет меняться по мере роста доступности и популярности технологии энергонезависимой памяти (Non-Volatile Memory, NVM) [ARULRAJ17]. Энергонезависимое хранилище уменьшает или полностью устраняет (в зависимости от конкретной технологии) асимметрию между временем чтения и записи, дополнительно улучшает производительность чтения и записи и обеспечивает доступ с байтовой адресацией.

---

[1] Сравнение задержек доступа к диску и памяти и многих других важных параметров, приведенных за несколько лет, и их графическое представление см. на https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html.

Долговечность в резидентных хранилищах

Резидентные СУБД сохраняют резервные копии на диске, чтобы обеспечить долговечность и не допустить потери часто меняющихся данных. Хотя некоторые базы данных хранят данные исключительно в памяти, без каких-либо гарантий долговечности, мы не будем обсуждать их в рамках этой книги.

Прежде чем операция будет считаться завершенной, ее результаты должны быть записаны в последовательно организованный журнал предзаписи. Более подробно мы рассмотрим журналы упреждающей записи в разделе «Восстановление» на с. 106. Чтобы избежать применения полного содержимого журнала во время запуска или после сбоя, резидентные хранилища поддерживают *резервную копию*. Резервная копия поддерживается в виде упорядоченной структуры на диске; при этом изменения часто вносятся в нее асинхронно (без привязки к запросам клиентов) и применяются пакетами для уменьшения числа операций ввода-вывода. Во время восстановления содержимое базы данных может быть получено из резервной копии и журналов.

Записи журнала обычно применяются к резервной копии пакетами. После обработки пакета записей журнала резервная копия содержит *моментальный снимок* базы данных, который соответствует некоторому моменту, и все предшествующее содержимое журнала может быть удалено. Этот процесс называется *созданием контрольных точек*. При таком подходе сокращается время восстановления за счет поддержания дисковой базы данных в предельно актуальном состоянии с помощью записей журнала, без необходимости в блокировке доступа клиентов на время обновления резервной копии.

Не стоит думать, что резидентная база данных представляет собой что-то вроде дисковой базы данных с огромным страничным кэшем (см. раздел «Организация буферизации данных» на с. 98). При кэшировании страниц в памяти формат сериализации и способ представления данных несут с собой дополнительные издержки и не позволяют достичь той же степени оптимизации, которую могут обеспечить резидентные хранилища.

Дисковые базы данных используют специализированные структуры хранения данных, оптимизированные для доступа к диску. Поскольку в памяти сравнительно быстро осуществляются переходы по указателям, произвольный доступ к памяти осуществляется намного быстрее, чем произвольный доступ к диску. Дисковые структуры хранения данных часто имеют вид широкого и низкого дерева (см. подраздел «Деревья для дисковых хранилищ», с. 46), в то время как резидентные хранилища могут использовать больше разновидностей структур данных и способны на такие виды оптимизации, которые невозможно или очень сложно реализовать на диске [MOLINA92]. Точно так же на диске особого внимания требует обработка данных переменного размера, в то время как в памяти это обычно решается путем обращения к значению с помощью указателя.

В случае некоторых сценариев использования уместно предположить, что весь набор данных поместится в памяти. Некоторые наборы данных ограничены природой

вещей реального мира, которые они представляют: записи учащихся для школ, записи клиентов для корпораций или товарные запасы в интернет-магазине. Каждая запись занимает не более нескольких КБ, а их количество ограничено.

Колоночные и строчные СУБД

Большинство СУБД хранит некоторый *набор записей*, состоящий из *столбцов* и *строк таблиц*. *Поле* находится на пересечении столбца и строки и содержит одно значение некоторого типа. Поля, относящиеся к одному столбцу, обычно имеют один и тот же тип данных. Например, если мы определим таблицу, содержащую записи пользователей, все имена будут иметь один и тот же тип и находиться в одном и том же столбце. Набор значений, логически относимых к одной и той же записи (обычно идентифицируемой ключом), образует строку.

Один из способов классификации баз данных сводится к их разделению в зависимости от того, как данные сохраняются на диске: по строкам или по столбцам. Данные таблиц могут разбиваться либо по горизонтали (когда вместе сохраняются значения, относящиеся к одной строке), либо по вертикали (когда вместе сохраняются значения, относящиеся к одному столбцу). Разница между этими способами показана на рис. 1.2: (*а*) значения разбиваются по столбцам и (*б*) значения разбиваются по строкам.

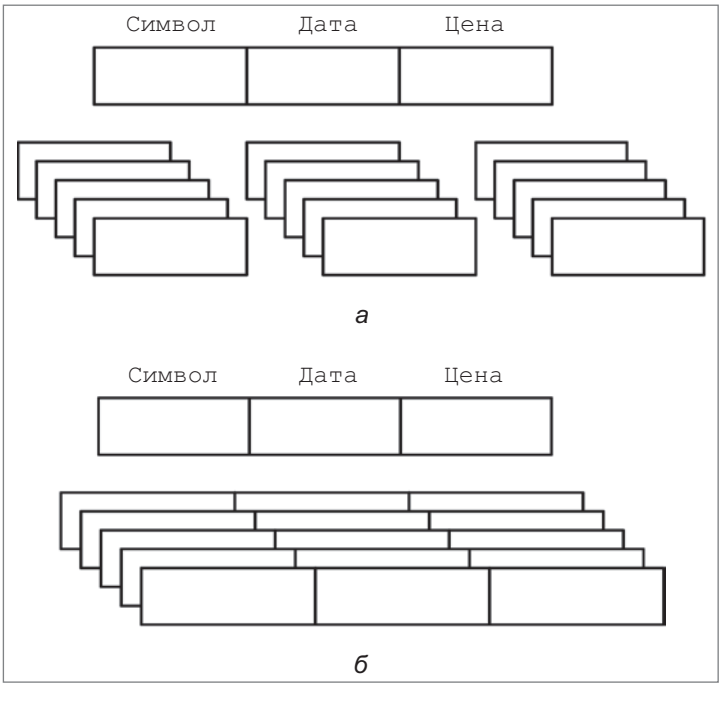

Рис. 1.2. Компоновка данных в колоночных и строчных хранилищах