СПбГЭТУ «ЛЭТИ»
ПЕРВЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ

Татарникова Татьяна Михайловна

Управление большими данными

Конспект лекций

СПбГЭТУ «ЛЭТИ», 2022 г.

ТЕМА 1. БАЗЫ ДАННЫХ NOSQL

Базы данных NoSQL появились в ответ на необходимость обрабатывать «большие» данные на крупных аппаратных платформах, состоящих из вычислительных кластеров. Таким образом, в настоящее время реляционные БД уже не являются единственной моделью отображения данных. У разработчиков информационных систем появился выбор модели хранения и управления данными.

1.1. Нереляционная модель данных

За то время, пока реляционная модель была практически единственной при отображении физической модели хранения данных, накопилось ряд проблем, которые реляционные БД решают неэффективно или даже неестественно.

Вот некоторые из них.

1. Для реляционных баз данных характерна потеря соответствия. Проявляется она в том, что реляционные базы данных не позволяют хранить агрегаты. Агрегат – термин, пришедший из предметно-ориентированного проектирования, где агрегатом называют коллекцию связанных объектов, которая интерпретируется как единое целое, что-то наподобие коробки, которую мы подписываем и складываем в нее предметы, которые должны находиться вместе.

Например, чтобы отобразить всю информацию о покупателе и всех его покупках необходимо собрать данные из многих таблиц: покупатель, заказ, пункт заказа, цена и др. и хранить их в виде соответствия. При значительном росте числа таблиц формирование агрегата покупателя существенно осложняется.

2. С увеличением объема хранимых данных возникает задача фрагментации таблиц базы данных по разным серверам, объединенных в кластер. При выполнении сложных запросов производительность системы существенно уменьшается в результате межмашинного обмена данными между серверами кластера. Возникает проблема обеспечения надежности кластера.

3. Схема базы данных состоит из подсхем, каждая из которых отражает предметную область какого-либо подразделения организации. Модификация подсхемы одного подразделения и реорганизация соответствующих таблиц приводит к вынужденной приостановке работы остальных подразделений. Поддержка целостности данных и оперативности такого взаимодействия является сложной задачей.

Как попытка решить накопившиеся проблемы реляционных баз данных появились альтернативные средства хранения и обработки данных, получившие название «базы данных NoSQL». Пионерами в этой области выступили две компании: Google и Amazon,

Идея нереляционных баз данных очень проста: данные хранятся в виде записей <ключ, значение>, а схема базы данных отсутствует. При этом, в поле «значение» может храниться агрегат, например, вся информация о покупателе и его покупках. Так решается первая проблема, присущая реляционным БД.

Данные в виде агрегатов автоматически равномерно распределяются и реплицируются по серверам кластера. Таким образом решается вторая проблема, присущая реляционным БД.

Отсутствие схемы базы данных позволяет включать или удалять атрибуты на уровне отдельной записи, не затрагивая работу остальной части системы, что решает третью проблему, свойственную реляционным БД.

Термин «NoSQL» в настоящее время не имеет строгого определения. И не существует авторитетного органа, который бы предложил такое определение, поэтому можно говорить о некоторых общих свойствах БД, относящихся к категории NoSQL.

Первое свойство – очевидный факт: базы данных NoSQL не используют язык SQL. Некоторые из них имеют свой язык запросов, многие из которых похожи на SQL чтобы их легче было изучить. Однако до сих пор не реализован ни один язык, который достиг той же степени гибкости, что и стандартный язык SQL.

Другое важное свойство этих баз данных заключается в том, что они представляют собой проекты с открытым исходным кодом. Несмотря на то что термин «NoSQL» часто применяется к системам с закрытым исходным кодом, существует мнение, что NoSQL – это феномен с открытым исходным кодом.

Большинство баз данных NoSQL создавались в ответ на необходимость работать на кластерах. Для обеспечения согласованности в реляционных БД используют транзакции. Это изначально противоречит кластерной среде, поэтому базы данных NoSQL предлагают спектр вариантов для обеспечения согласованности и распределения данных.

Базы данных NoSQL учитывают емкость веб-сайтов начала XXI века, поэтому обычно только системы, разработанные примерно в это время, называются NoSQL, тем самым исключая базы данных, созданные в прошлом веке.

Базы данных NoSQL работают без схемы, позволяя свободно добавлять поля в базу данных без предварительного изменения структуры. Это очень важно для БД с неоднородными данными

Все свойства, указанные выше, являются общими для баз данных NoSQL. Ни одно из них нельзя считать определяющим.

Таким образом сформулируем: базы данных NoSQL – это распределенные нереляционные базы данных с открытым исходным кодом. Из известных баз данных NoSQL можно назвать Cassandra, Riak, DynamoDB, Hypertable, MongoDB и другие.

Все БД NoSQL являются неструктурированными. Когда данные хранятся в реляционной базе, то сначала определяется схема БД. В базах данных NoSQL хранение данных происходит по-другому.

Каждая БД, реализованная по технологии NoSQL использует свою собственную модель. Эти модели разделяются на четыре категории:

– ключ-значение;
– документ;
– семейство столбцов;
– граф.

БД типа «ключ-значение» позволяет хранить данные по ключу.

Документная база данных по существу делает то же самое, поскольку не накладывает ограничений на структуру хранящихся документов.

Семейство столбцов позволяет хранить любые данные в любом столбце.

Графовые базы данных отображают объекты и связи между ними в виде графа. Путем добавления, удаления, изменения ребер и узлов графа происходит управление данными.

Первые три модели объединяет свойство агрегатной ориентацией. Рассмотрим, что понимают под агрегатами и как они влияют на модели данных.

Реляционная модель хранимую информацию разделяет на кортежи (строки). Кортеж – это ограниченная структура данных. Он хранит набор значений, поэтому не может содержать запись, список значений или другой кортеж. Эта простота образует основу реляционной модели и позволяет интерпретировать все операции как операции над кортежами и возвращение кортежей.

Агрегатная ориентация придерживается другого подхода. Она учитывает необходимость оперировать данными, имеющими более сложную структуру, чем набор кортежей. Агрегат можно сравнить со сложной записью, которая может содержать списки и другие структуры записей (см. рис. 1.1). Агрегат не имеет строгого шаблона и в зависимости от задачи может иметь структуру разной сложности. Таким образом, агрегат представляет собой единицу для манипулирования данными и управления их согласованностью.

Рисунок 1.1 – Структура агрегата

Модификация агрегатов происходит с помощью атомарных операций. Взаимодействие с БД, как хранилищем данных выполняется посредством агрегатов.

Агрегаты облегчают работу баз данных на кластерах, поскольку представляет собой естественную единицу репликации и фрагментации. Кроме того, агрегаты упрощают разработку прикладных программ, которые часто манипулируют данными с помощью агрегированных структур.

Продемонстрируем сказанное на примере. Предположим, что разрабатывается веб-сайт для электронной торговли и планируется продавать товары непосредственно клиентами через web, что требует хранения информации о пользователях, каталогах товаров, заказы, адреса поставки и даты платежей. Этот сценарий используем для моделирования данных с помощью реляционной модели, а также с помощью технологии NoSQL, что позволит проанализировать их преимущества и недостатки.

Разработку реляционной базы данных можно начать с модели данных, представленной на рисунке 1.2. В ней выполнены все правила реляционной модели и проведена нормализация отношений.

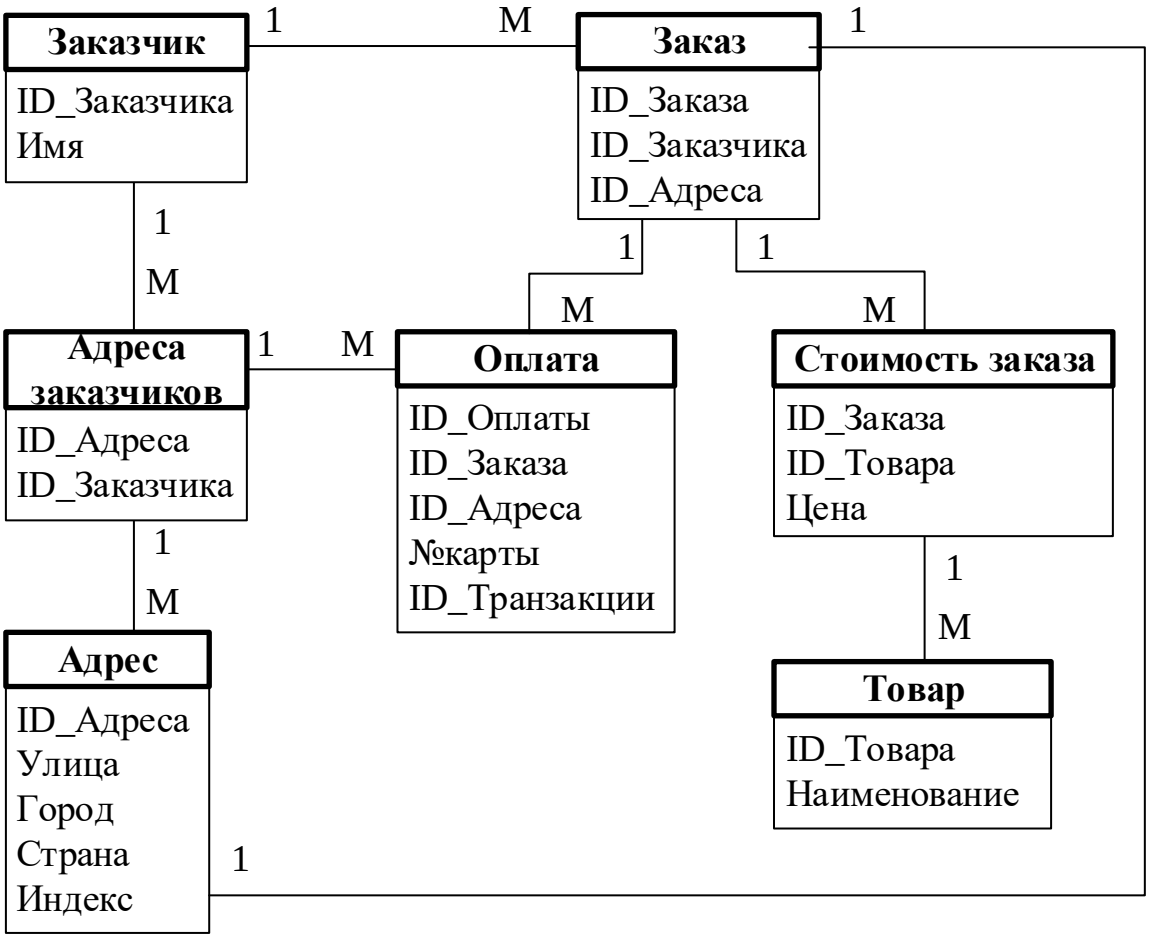

Рисунок 1.2 – Модель данных

Реляционная БД, соответствующая модели на рисунке 1.2 включает 7 отношений. На рисунке 1.3 приведены фрагменты некоторых из них.

Заказчик

ID_Заказчика	Имя
1	Яковлев С.А.
...	...

Заказ

ID_Заказа	ID_Заказчика	ID_Адреса
99	1	77
...	...	...

Адреса заказчиков

ID_Адреса	ID_Заказчика
77	1
...	...

...

Товар

ID_Товара	Наименование
27	«NoSQL»
...	...

Рисунок 1.3 – Фрагменты таблиц, являющихся частью реляционной БД

Посмотрим, как будет выглядеть эта модель, если применить агрегатно-ориентированный подход. Модель изобразим средствами UML-диаграммы (см. рис. 1.4). В UML-диаграмме ромб обозначает агрегацию.

Данные будем представлять в формате JSON[1], который является основным способом представления данных в технологии NoSQL. В модели есть два основных агрегата: **Заказчик** и **Заказ**.

Агрегат «**Заказчик**» содержит список адресов заказчиков.

Агрегат «**Заказ**» содержит список заказанных товаров, адреса поставки и данные о платежах. Запись о платеже сама содержит адрес заказчика, выполняющего данный платеж.

Отдельная логическая запись, содержащая адрес, в этом примере появляется три раза, но вместо использования идентификатора она интерпретируется как значение и каждый раз копируется. При использовании агрегатов можно копировать всю адресную структуру в агрегат.

Название товара показано в качестве части заказа, – этот вид денормализации напоминает компромисс, принятый в реляционных базах данных, но по отношению к агрегатам он носит более общий характер, потому что есть необходимость минимизировать количество агрегатов, к которым будет осуществляться доступ при работе с данными

[1] JSON – текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми

Связь между заказчиком и заказом не хранится в агрегатах. Аналогично связь, идущая от заказа, может идти к отдельной агрегированной структуре для товаров, но она не хранится в этой структуре.

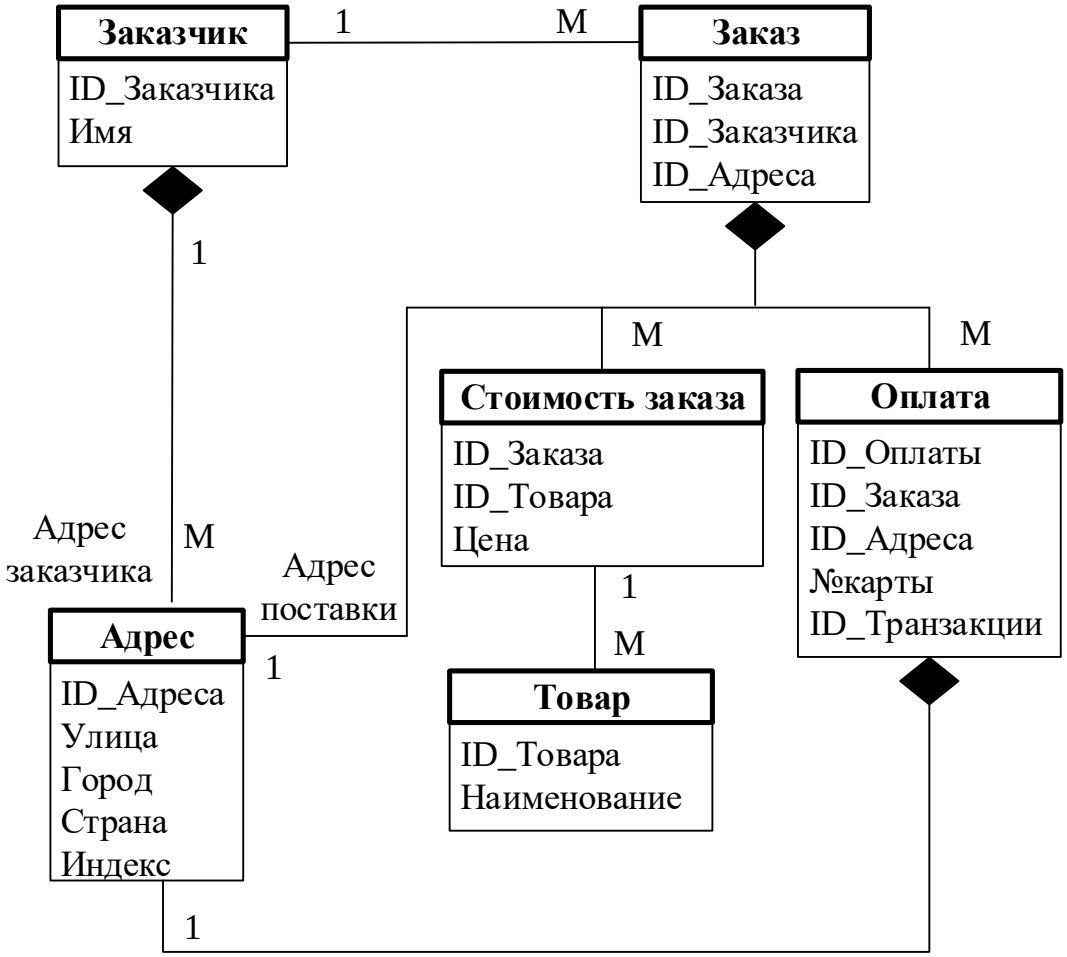

Рисунок 1.4 – Агрегатная модель данных

// **Заказчик**

```
{ «ID_Заказчика":1,
  "Имя":"Яковлев С.А.",
  "Адрес":
       [
       {"ID_Адрес":" 55",
        "Улица":" Б. Морская",
        "Город": "Санкт-Петербург",
        "Страна":"Россия",
```

"Индекс": "190000"
}
],
}
//**Заказ**
{"ID_Заказа":99,
" ID_Заказчика":1,
"Стоимость заказа": [
{ "ID_Товара":27,
"Цена": 320,
"Наименование": "NoSQL"
}
] ,
"ID_Адрес": [{ "ID_Адреса": "55"}],
"Оплата": [
{ "№карты":"1000-1000-1000-1000",
"ID_транзакции":"abelif879rft",
"ID_Адреса": {"ID_Адреса": "55"}
}
],
}

В данном примере важен не столько конкретный способ изображения границы агрегата, сколько тот факт, что необходимо думать о доступе к данным при разработке модели данных для приложения.

Действительно, можно иначе изобразить границы агрегатов, поместив все заказы отдельного заказчика в агрегат «Заказчик», как на рисунке 1.5.

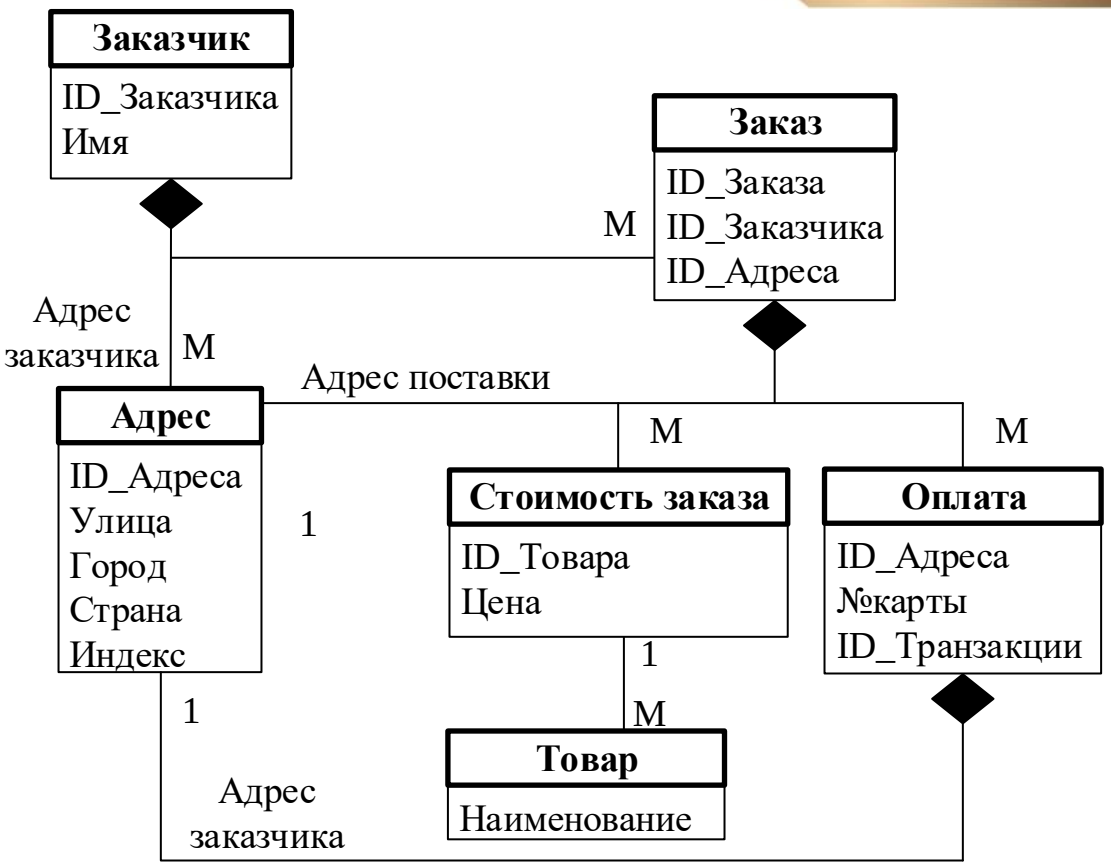

Рисунок 1.5 – Агрегатная модель, в которой все объекты объединены в один агрегат «Заказчик»

//Заказчик

{«ID_Заказчика":1,
"Имя":"Яковлев С.А.",
"Адрес": [
{"ID_Адрес":" 55“,
"Улица":" Б. Морская",
"Страна":"Россия",
"Индекс": "190000"
}],
"Заказ": [
{"ID_Заказа":" 99",
"Стоимость заказа":[
{“ID_Товара":27,

"Цена": "300 “
}
]
}
],
"Оплата":[
{"№карты":"1000-1000-1000-1000",
"ID_транзакции":"abelif879rft",
"ID_Адреса": {"ID_Адреса": "55"}
}]
}

Универсального способа для изображения границ агрегатов не существует. Это целиком зависит от целей манипулирования данными. Например, чтобы отобразить всю информацию о заказчике и всех его заказах, программист должен собрать в оперативной памяти данные из многих таблиц: заказчик, заказ, адрес заказа, цена и др. При значительном росте числа таблиц разработчик часто просто забывает назначение той или иной таблицы, осложняется связывание таблиц при выполнении запроса, то есть существенно осложняется формирование агрегата. Поэтому, если необходимо получать доступ к записи о заказчике и ко всем его заказам одновременно, то, вероятно, предпочтительнее один агрегат. Однако, если необходимо в каждый момент времени получать доступ к отдельному заказу, то лучше предусмотреть отдельный агрегат для каждого заказа. Естественно, это сильно зависит от данных; даже в рамках одной системы разные приложения могут иметь разные предпочтения.

Большинство баз данных NoSQL не поддерживают механизм транзакций. Известно, что транзакции – это механизм, который обеспечивают согласованность данных. Но в БД NoSQL поддерживаются другие механизмы манипуляции с отдельными агрегатами по очереди из кода приложения.

Итак, обобщая информацию об агрегатной (нереляционной) модели данных можно сделать следующие выводы:

1. В агрегатной (нереляционной) модели данных существует неявная схема, подразумеваемая программистом БД.

2. Границы агрегатов выбираются программистом БД и во многом зависят от целей манипулирования данными.

3. Агрегаты объединяют в одно целое данные, доступ к которым осуществляется одновременно.

4. Агрегаты указывают, какие части данных должны храниться на одном и том же узле.

5. Если реляционные базы данных позволяют манипулировать любой комбинацией строк из любой таблицы в рамках одной транзакции, то в нереляционных БД аналогичная задача решается разделением данных по агрегатам.

1.2. Распределение и согласованность

Основным свойством технологии NoSQL является возможность функционирования баз данных на большом кластере, то есть размещение базы данных на кластере серверов. Этот процесс также называется горизонтальным масштабированием базы данных.

Агрегатно-ориентированный подход хорошо согласуется с горизонтальным масштабированием, поскольку агрегат является естественной единицей распределения. В зависимости от выбранной модели распределения можно создать хранилище данных, предоставляющее возможность обрабатывать больший объем данных, обрабатывать более интенсивный трафик чтения или записи, а также избегать перегрузки и торможения компьютерной сети.

С другой стороны, работа на кластерах повышает сложность базы, поэтому при выборе модели распределения взвешиваются все аргументы за и против.

Существуют два способа распределения данных: репликация и фрагментация (см. рис. 1.6).

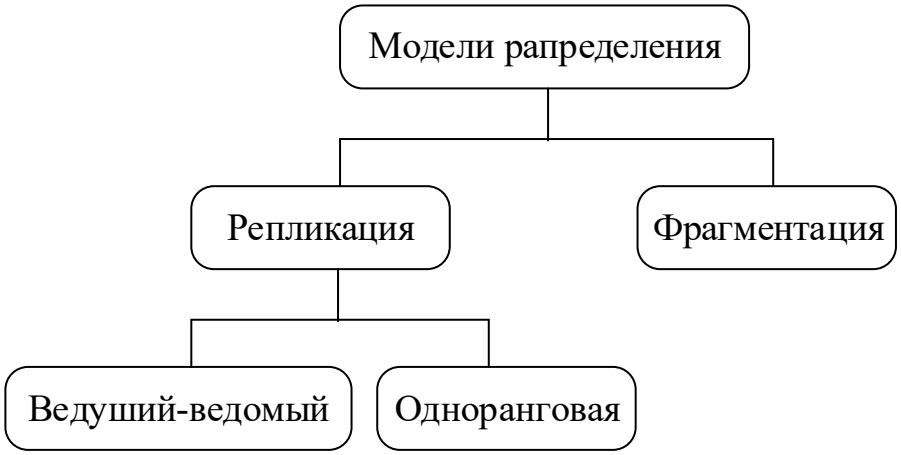

Рисунок 1.6 – Классификация способов распределения данных

Репликация подразумевает копирование одних и тех же данных на нескольких узлах (реплик). Количество реплик называется коэффициентом репликации. Репликация бывает двух видов: ведущий-ведомый и одноранговая.

Фрагментация подразумевает, что разные части БД размещаются на множество серверов. Если механизм фрагментации не применять, то пользователи будут обращаться к одному большому серверу. При наличии механизма фрагментации пользователи будут обращаться к разным узлам и благодаря этому получать быстрые ответы. Например, если БД распределена в кластере из 10 серверов, то каждый из них будет загружен только на 10%. Разумеется, идеальный случай крайне редок. Для того чтобы приблизиться к идеалу, необходимо, чтобы данные, которые запрашиваются одновременно, размещались вместе на одном и том же узле для ускорения доступа к ним. Поэтому актуальность вопроса заключается в том, как сгруппировать данные так, чтобы один пользователь в основном получал данные с одного сервера. Это тот случай, когда помогает агрегатная ориентация, т.к. главное свойство агрегата заключается в том, что он объединяет данные, которые, как правило, запрашиваются одновременно. Таким образом, агрегаты являются естественной единицей распределения.

Репликация и фрагментация являются ортогональными методами: можно использовать любую из двух или обе вместе.

Рассмотрим эти методы подробнее.

При **распределении по схеме «ведущий-ведомый»** происходит репликация данных по многим узлам. Один узел назначается ведущим (master), или главным. Этот ведущий узел является доверенным источником данных и обычно несет ответственность за выполнение всех модификаций этих данных. Остальные узлы являются ведомыми (slaves), или вторичными. Процесс репликации синхронизирует ведомые узлы с ведущим.

Репликация «ведущий-ведомый» – это решение для тех баз данных, к которым интенсивно выполняется операция чтения. То есть, чтобы выполнить больше запросов на чтение, необходимо добавить больше ведомых узлов и направлять на них все запросы на чтение. Однако если обновлять данные на ведущем узле, есть опасность, что чем больше ведомых узлов, тем выше вероятность, что пользователи будут получать несогласованные данные. То есть это неудачное решение для баз данных с интенсивным трафиком записи.

Второе преимущество репликации «ведущий-ведомый» – это отказоустойчивость чтения, то есть, если на ведущем узле произойдет отказ, ведомые узлы смогут по-прежнему обрабатывать запросы на чтение. Сбой ведущего узла сделает запись невозможной, пока его работа не будет восстановлена или не будет подключен новый ведущий узел. Как раз наличие реплик ведущего узла на ведомых узлах ускоряет процесс его восстановления после сбоя.

Таким образом, репликация имеет не только привлекательные свойства, но и неизбежный недостаток – несогласованность. Существует опасность, что разные клиенты, читающие данные с ведомых узлов, получат разные значения из-за того, что обновления не успеют распространиться по всем ведомым узлам.

По существу, ведущий узел остается узким местом и единственной точкой отказа в модели «ведущий-ведомый».

Одноранговая репликация решает эту проблему, устраняя ведущий узел. Все реплики имеют одинаковый вес, все могут выполнять операции записи, и потеря любой из них не приводит к потере доступа к хранилищу данных.

Самая большая сложность, как и раньше, – согласованность. Когда выполняется запись в два разных места, есть риск, что два человека попробуют обновить одну и ту же запись в один и тот же момент времени. Таким образом возникает конфликт «запись-запись». Несогласованность чтения тоже приводит к проблемам, но они являются преодолимыми. А вот, несогласованность записи имеет необратимый характер.

Одним из глобальных отличий реляционной базы данных от базы данных NoSQL – это то как обеспечивается согласованность данных.

Согласованность – это обеспечение внутренней непротиворечивости данных. Различают строгую и итоговую согласованность. Строгая гарантирует, что данные вернутся полностью достоверными и не устареют. Итоговая согласованность не может гарантировать, что данные вернуться полностью достоверными, но в итоге данные обновятся на всех репликах.

Согласованность проявляется в разных формах. Рассмотрим примеры.

Представим процедуру **обновления данных**, при котором случайно два пользователя, назовем их User1 и User2 внесли обновления. Эта проблема называется конфликтом «запись-запись». Она возникает, когда два пользователя БД обновляют одни и те же данные в один и тот же момент времени *tw* (см. рис. 1.7). Когда записи достигают сервера, тот их сериализирует – обрабатывает одну, а

потом другую, например, в алфавитном порядке, и реализует сначала обновление первого пользователя User1, а затем обновление User2. Если контроля согласованности нет, то обновление первого пользователя будет выполнено, а затем немедленно перекрыто обновлением второго пользователя. В этом случае обновление первого пользователя называется потерянным. Это явление можно считать нарушением **согласованности обновления**, потому что обновление второго пользователя использовало состояние до обновления первого, но применялось после него.

Репликация повышает вероятность конфликтов «запись-запись». При обновлении реплик на разных узлах независимо друг от друга необходимы специальные меры обеспечения согласованности данных. Решение, которое способствует снижению вероятности возникновения конфликта «запись-запись» заключается в том, чтобы все записи определенных данных хранить на одном узле. Это решение использовалось во всех распределенных моделях, рассмотренных выше, кроме одноранговой репликации.

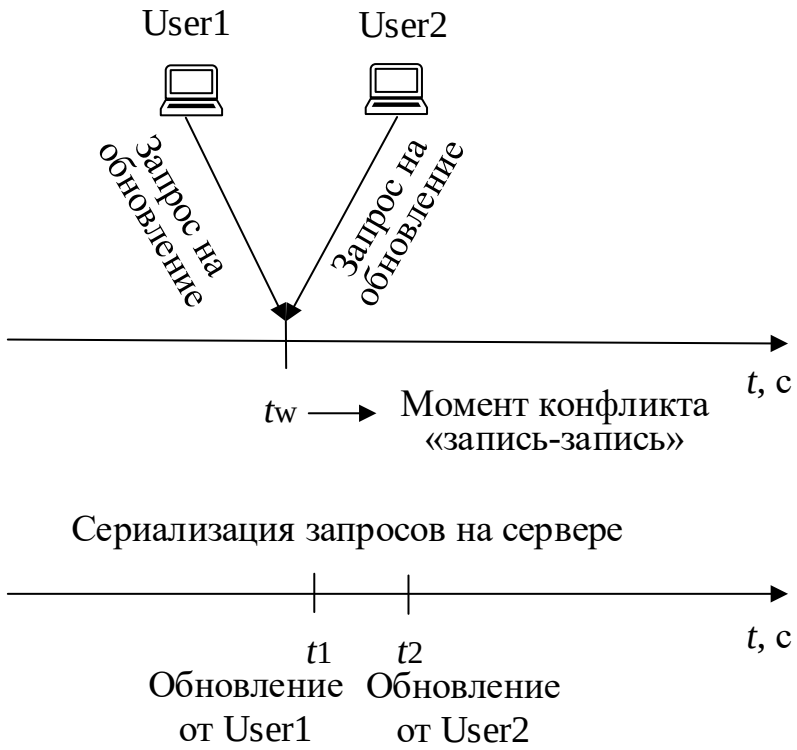

Рисунок 1.7 – Конфликт обновления данных

Представим процедуру чтения данных. Пусть оформляется заказ на определенные товары с определенными расходами на доставку. Расходы на доставку рассчитываются на основе товаров, указанных в заказе. Если добавляется новая

позиция, то вычисления необходимо выполнить заново и обновить запись о расходах на поставку.

Опасность несогласованности заключается в том, что первый пользователь сначала добавляет позицию в свой заказ, а затем второй пользователь считывает эти позиции и расходы на доставку, а после этого первый пользователь обновляет запись о расходах на доставку. Этот вид ошибки называется **несогласованным чтением** или конфликтом «чтение-запись».

На рисунке 1.8 показана ситуация, в которой второй пользователь выполнил чтение в середине процедуры записи, выполняемой первым пользователем. Для исключения такой ситуации используется метод логической согласованности. Она гарантирует, что разные элементы данных будут изменяться вместе. Например, для того чтобы избежать логической несогласованности при конфликте «чтение-запись», реляционные базы данных используют понятие транзакций. Обе записи первого пользователя упаковываются в одну транзакцию, и тем самым система гарантирует, что второй пользователь будет читать оба элемента данных либо до, либо после обновления.

Агрегатная модель базы данных NoSQL позволяет избежать несогласованности сделав заказ, стоимость доставки и товарные позиции заказа частями одного и того же агрегата.

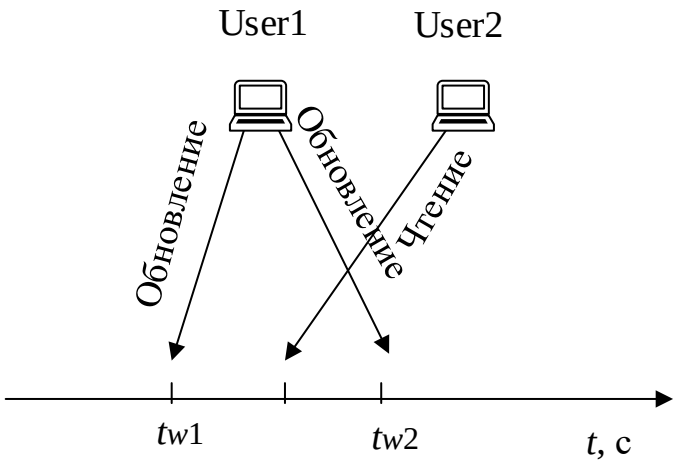

Рисунок 1.8 – Конфликт чтения данных

Разумеется, не все данные можно записать в один и тот же агрегат, поэтому любые обновления, влияющие на несколько агрегатов, оставляют интервал времени, в течение которого клиенты могут выполнить несогласованное чтение.

Продолжительность этого интервала называется окном несогласованности. Система NoSQL может иметь довольно узкое окно несогласованности. Например, в документации компании Amazon сказано, что окно несогласованности обычно не превышает секунды.

С появлением репликации появился и новый вид несогласованности. На рисунке 1.9 приведено пояснение этого вида несогласованности.

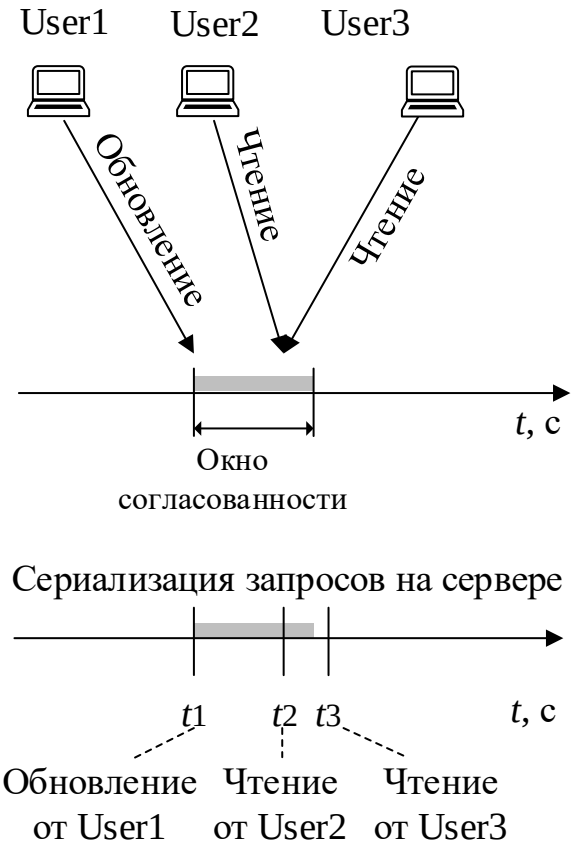

Рисунок 1.9 – Нарушение согласованности репликаций

Пользователь User1 вносит обновления, но наличие окна несогласованности означает, что разные пользователи БД увидят разные данные в одно и то же время. Поэтому User2 получит недостоверные данные (еще не обновленные), а пользователь User3 – уже обновленные.

Таким образом, согласованность репликаций гарантирует, что при чтении с разных реплик один и тот же элемент данных имеет одно и то же значение.

Разумеется, в конце концов, обновления будут полностью распределены по узлам, и User2 увидит обновленные данные. По этой причине такая ситуация называется итоговой согласованностью или согласованностью «в конечном

счете». Это значит, что в любой момент времени узлы могут быть несогласованными, но, если нет новых обновлений, в конце концов все узлы будут обновлены и получат одно и то же значение.

Согласованность – это важное свойство БД, но, к сожалению, иногда ею приходится жертвовать. Почти всегда можно разработать систему, предотвращающую несогласованность, но практически невозможно сделать это без ущерба для остальных характеристик системы. В результате часто приходится жертвовать согласованностью в пользу чего-то другого.

В NoSQL этот компромисс формулируется теоремой CAP. CAP – это акроним от трех свойств:

– **Согласованность** (**C**onsistency) – непротиворечивость данных.
– **Доступность** (**A**vailability) – предельное время отклика, которое допустимо.
– **Устойчивость к разделению** (**P**artition tolerance) означает, что кластер может восстанавливать обмен данными после обрыва связей в кластере, который разделен на многочисленные фрагменты, не способные взаимодействовать друг с другом

Основное утверждение этой теоремы гласит, что из трех свойств – согласованности данных, доступности и устойчивости к разделению – можно обеспечить не больше двух. На практике CAP реализует следующее утверждение: в системе, которая подвержена разделению, следует искать компромисс между согласованностью и доступностью. Полученная в результате система не будет ни хорошо согласованной, ни идеально доступной, но она будет представлять собой разумное сочетание этих свойств.

Чем больше узлов задействовано в запросе, тем выше вероятность возникновения конфликта «запись-запись» и ниже вероятность конфликта «чтение-запись». Отсюда естественен вопрос: «Сколько узлов (реплик) должно быть вовлечено в запрос, чтобы обеспечить строгую согласованность данных?»

Введем следующие обозначения:

R – коэффициент репликации;

W – количество узлов, участвующих в записи;

N – количество узлов, участвующих в чтении.

Согласно теореме CAP, не нужно, чтобы все узлы подтверждали запись для обеспечения строгой согласованности; нужно, чтобы это сделали большинство. Это явление называется кворумом записи и выражается неравенством

$$W>N/2 \quad (1.1)$$

Аналогично существует кворум чтения, но это более сложное понятие. Кворум чтения зависит от того, сколько узлов должны подтвердить запись. Пусть R=3. Если все записи должны подтвердить два узла (W=2), то необходимо установить контакт по крайней мере с двумя узлами, чтобы гарантировать получение последних данных. Если же записи подтверждаются только одним узлом (W=1), надо связаться со всеми тремя узлами, чтобы гарантировать получение последних обновлений. В последнем случае нет кворума записи, поэтому возникает конфликт обновлений, но, контактируя с достаточно большим количеством читателей, обнаружение конфликта гарантированно. Таким образом, можно получить строго согласованные результаты чтения, даже если нет строгой согласованности записей. Неравенство получения строгой согласованности:

$$R+W>N \quad (1.2)$$

Неравенства (1.1) и (1.2) выведены для одноранговой модели распределения.

В случае распределения «ведущий-ведомый», чтобы избежать конфликтов «запись-запись», достаточно записать данные на ведущий узел. Чтобы избежать конфликтов «чтение-запись», достаточно выполнять чтение только с ведущего узла. Уточним, что количество узлов в кластере и коэффициент репликации – это разные числа. Например, при фрагментации баз данных кластер можете иметь 100 узлов при коэффициенте репликации, равном 3.

Обобщим информацию о правилах распределения и согласованности данных в следующих выводах:

1. Конфликты «запись-запись» возникают, когда два клиента пытаются записать одни и те же данные в одно и то же время.

2. Конфликты «чтения-записи» в распределенных системах возникают, когда некоторые узлы получают обновленные данные, а другие нет.

3. Итоговая согласованность означает, что определенная часть системы станет согласованной, как только все записи будут распространены по всем узлам.

4. Чтобы обеспечить хорошую согласованность данных в распределенных системах возникает необходимость компромисса между согласованностью и временем реакции (теорема CAP).

1.3. Базы данных «ключ – значение»

Базы данных «ключ – значение» состоит из множества агрегатов, каждый из которых имеет ключ или идентификатор, который используется для доступа к данным. Агрегаты в такой базе данных являются непроницаемыми, то есть просматривать содержимое агрегата можно только с помощью его ключа.

Внешне БД «ключ – значение» похожа на реляционную БД с двумя столбцами, например, ID и Имя (см. табл. 1.1), где столбец ID – первичный ключ, а столбец Имя содержит значение. При этом значение – это двоичный объект данных, который записан в хранилище без детализации его внутренней структуры в виде хэш-кода. Это позволяет с одной стороны избавиться от метаданных, то есть хранить прямое значение, а с другой стороны обеспечить целостность данных. Что именно хранится в этом объекте может определить только приложение.

Табл. 1.1. БД «ключ – значение»

ID	Имя
AAAAA	1001010100010001…
AABAB	0100010110011101…
DFA766	0000111101101101…
FABCC4	1100110010101110…

Из-за хранения данных в поле значение в виде хэш-кода говорят, что БД «ключ-значение» – это хэш-таблица. Выбор функции хэширования должен обеспечить равномерное распределение хэшированных ключей по хранилищу данных.

С точки зрения интерфейса прикладного программирования хранилище типа «ключ-значение» – самое простое из БД NoSQL. Управление данными в плане манипулирования ими сводится к тому, что клиент может либо получить значение по ключу, либо записать значение по ключу, либо удалить ключ из хранилища данных. Поскольку хранилища типа «ключ-значение» всегда используют

доступ по первичному ключу, они обычно имеют высокую производительность и легко масштабируются.

Приложение вычисляет ID и значение и сохраняет эту пару. Если ключ ID уже существует, то текущее значение замещается, в противном случае создается новая запись.

Если есть необходимость хранить данные сеанса пользователя, информацию о его корзине товаров и предпочтениях, то можно записать их в один агрегат с одним ключом для всех перечисленных объектов (см. рис. 1.10).

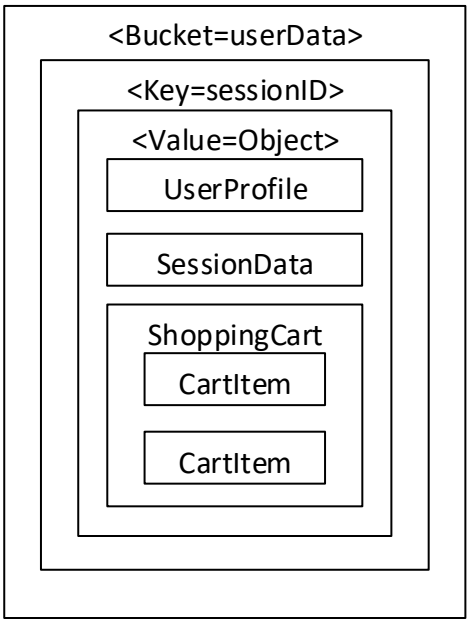

Рисунок 1.10 – Пример агрегата БД «ключ-значение»

Недостатком хранения всех объектов в одном агрегате является тот факт, что агрегаты могут иметь разные типы, которые могут вызвать конфликты ключей.

В качестве альтернативы к ключу можно было бы добавить имя объекта, чтобы при необходимости можно было извлечь отдельный объект по этому имени (вложение ключ-значение), как на рисунке 1.11.

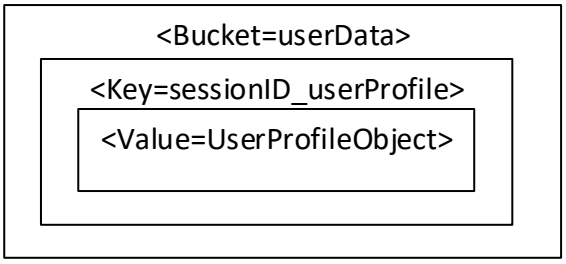

Рисунок 1.11 – Альтернативный вариант агрегата

Соответствие терминологии, принятой в реляционной модели БД и модели «ключ-значение» приведено в таблице 1.2.

Табл. 1.2. Соответствие терминов

Реляционная модель БД	Модель «ключ-значение»
База данных	Кластер
Таблица	Сегмент
Строка	Ключ-значение
Идентификатор строки	Значение

При использовании хранилищ типа «ключ-значение» большое внимание уделяется выбору структуры ключа – алгоритмам генерации ключа, использованию меток времени, индивидуальных данных пользователя в этих алгоритмах.

Благодаря своим характеристикам базы данных типа «ключ-значение» часто используют для хранения данных о пользовательских сессиях (при этом в качестве ключа используется идентификатор сессии), корзин покупателей, профилей пользователей и т.п.

Рассмотрим примеры, в которых хранилища типа «ключ-значение» на практике зарекомендовали себя с лучшей стороны:

Хранение информации о сессии.

Каждая веб-сессия является уникальной и имеет уникальный идентификатор SessionId. Приложение записывает идентификатор SessionID в БД и всю информацию о сессии одним запросом. Операции выполняются очень быстро, поскольку вся информация о сессии хранится в одном объекте.

Профили пользователей, предпочтения, товары.

Почти каждый пользователь имеет уникальный атрибут UserID, UserName или какой-то другой идентификатор, а также предпочтения, например, язык, цвет, часовой пояс, выбранные товары и т.д. Все это можно поместить в один объект и получать предпочтения пользователя с помощью одной операции. Аналогично можно хранить профили товаров.

Корзины заказа.

Коммерческие веб-сайты используют корзины заказа, связанные с пользователем. Если требуется, чтобы корзина заказа была доступна постоянно, независимо от браузеров, компьютеров и сессий, всю информацию о покупках можно поместить в объект Value с ключом UserId.

Существуют ситуации, в которых хранилища типа «ключ-значение» не являются оптимальным выбором. Приведем примеры таких ситуаций:

Запрос по данным.

В базах данных типа «ключ-значение» отсутствие структурированности данных не позволяет задать условия к конкретному внутреннему агрегату.

Отношения между данными.

Если между разными наборами данных необходимо установить отношения или поддерживать корреляцию между разными наборами ключей, то базы данных типа «ключ-значение» не имеют такой возможности.

Операции с множествами.

Поскольку операции в каждый момент времени ограничены одним ключом, невозможно работать с несколькими ключами одновременно. Если требуется обработать несколько ключей сразу, то это придется делать на клиентской стороне.

Транзакции, состоящие из многих операций.

Если при сохранении нескольких ключей при записи одного из них произошел сбой нет возможности вернуться в исходное положение или выполнить откат остальных операций.

Изучение особенностей баз данных типа «ключ-значение» позволяет сделать следующие выводы:

1. База данных «ключ-значение» – это хеш-таблица.
2. Доступ к данным БД реализуется только по ключу.
3. Для извлечения информации об отдельном объекте необходимо создать его уникальный ключ.
4. Объекты можно объединять в агрегаты.
5. База данных типа «ключ-значение» может быть распределенной (кластеры).
6. Базы данных типа «ключ-значение» имеют узкую область применения.

7. Базы данных типа «ключ-значение» не годятся для задач (запросов), типичных для реляционной модели БД.

1.4. Документные БД

Документные базы данных концептуально похожи на БД типа «ключ-значение», только в качестве значений хранятся документы. Таким образом, основной единицей манипулирования является документ, основная часть которого является неструктурированным текстом.

Документы хранятся примерно одинаково, но сами они не обязательно должны быть одинаковыми, то есть это тексты, графические, звуковые, мультимедийные документы.

Документные БД предназначены для создания, хранения и выдачи по запросам документов, содержащих требуемую информацию. В ответе на запрос к такой БД пользователь получает список документов, в определенной мере содержащих нужную ему информацию.

Мера соответствия полученного результата построенному запросу оценивается релевантностью. Релевантность – это характеристика, которая выражает полезность информации, относительно запроса, отправляемого в информационную систему.

Поисковые запросы к документным БД – это поиск смысловой (семантической) информации, например: выдать статьи, посвященные документным БД, то есть содержащие термин «документные БД».

Таким образом, поиск в документной БД сводится к сравнению смыслового содержания запроса со смысловым содержанием хранящихся в БД документов. Семантическая природа документов определяет необходимость учитывать синонимию, полисемию, омонимию, контекстную обусловленность смысла отдельного слова и возможность выразить один смысл многими способами.

База данных хранит и извлекает документы в форматах XML, JSON, jpeg, avi, wav и т.д.

Записью документальной базы данных является документ, который задается как набор необязательных полей, для каждого из которых определены имя и тип (см. рис. 1.12).

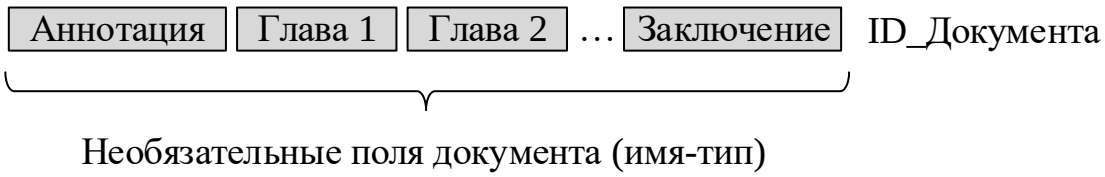

Рисунок 1.12 – Вид записи документальной БД

Для документных БД допустимы стандартные типы, задающие числовые, символьные и другие величины, но основной тип текстовый. Текстовые поля обладают переменной длиной и композиционной структурой, что не имеет прямых аналогов среди стандартных типов языков программирования. Текстовое поле состоит из параграфов, параграф – из предложений, предложение – из слов.

С точки зрения хранения, в документных БД идентифицируемым элементом данных будет поле, а с точки зрения поиска – слово.

Система управления документными базами данных присваивает каждому документу уникальный номер, а каждому ключевому слову документа ставится в соответствие указатель на списки экземпляров, являющихся перечнем документов, в которых встречается данное слово (то есть создается индекс). Каждый список экземпляров содержит заголовок, из которого можно узнать число экземпляров слова во всем файле документов, а также число документов, в которых это слово встречается.

Документальная БД включает в себя как минимум три области хранения данных, представляемые из-за своего большого размера, как правило, в виде файлов операционной системы (в действительности их всегда больше):

- файл словаря, устанавливающий соответствие между словом, встречающимся в БД, и его кодом;
- инверсный (инвертированный, обратный) список, содержащий для каждого слова БД список документов, его содержащих, используется при текстовом поиске;
- текстовый файл, содержащий собственно документы, используется при выдаче (просмотре) документов.

На рисунке 1.12 приведена принципиальная схема организации поиска документов, характерная для большинства современных документальных БД.

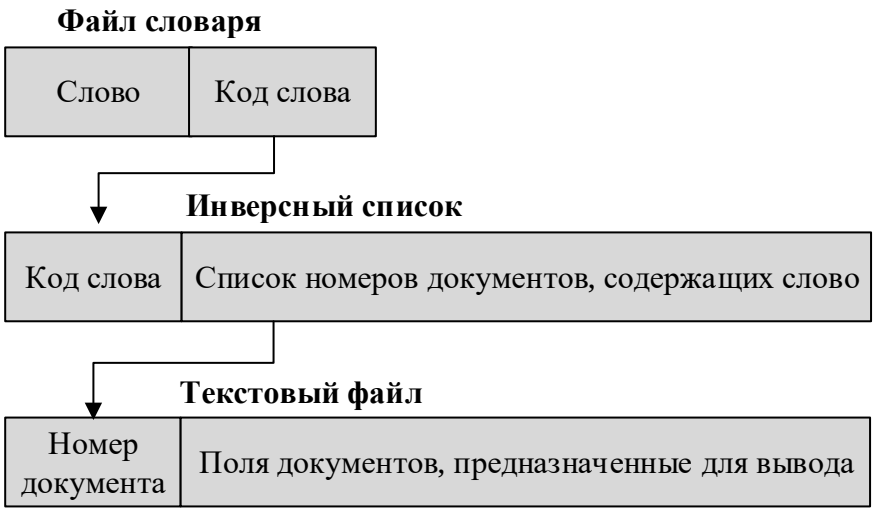

Рисунок 1.12 – Принципиальная схема организации поиска документов

Рассмотрим пример упрощенной реализации документальной БД в среде реляционной СУБД. С логической точки зрения она имеет «стандартную» структуру и включает две компоненты: регистрационные карты (РК) и полные тексты (ПТ).

Регистрационные карты представляют собой форматированные записи, содержащие относительно стандартный набор библиографических данных, а также ссылку на соответствующий полный текст (рис. справа).

Полные тексты документов состоят из страниц двух типов:

– логических, то есть структурных единиц текста – пунктов, параграфов, статьей;

– физических – фрагментов одинаковой длины, принудительно разбивающих длинный неструктурированный текст.

Организация физической структура документальной БД предполагает наличие следующих элементов:

Таблица ПТ – одна или несколько таблиц, в которых содержатся полные тексты документов. На логическом уровне образует представленную на рис.2 иерархическую структуру: БД, документ, страница.

Словарь ПТ – таблица представляет собой список ключевых слов и стандартных словосочетаний, извлеченных из текста, сопровождаемых частотами появления.

Инверсная таблица ПТ (или инверсный список ПТ) – таблица, содержащая список ключевых слов и словосочетаний, сопровождаемых номерами страниц.

Словарь и инверсная таблицы используются для сквозного полнотекстового поиска.

Таблица РК – таблица регистрационных карт, каждая запись которой содержит заглавие, дату регистрации, номер, вид документа, ссылки на страницы полного текста (ПТ) и другие поля.

Словарь РК – это таблица, содержащая значения полей регистрационных карт совместно с частотой появления и ссылками на записи таблицы РК.

Инверсная таблица РК (или инверсный список РК) содержит слова и словосочетания и ссылки на записи таблицы РК.

Словарная и инверсная таблицы используются для поиска записей РК, с последующим доступом к страницам полного текста.

Наряду со словарем РК иногда может использоваться словарь синонимов, служащий для обеспечения двуязычного поиска в словарных таблицах.

Благодаря такой организации физической структуры документной БД можно выполнять поиск двух видов:

- поиск по регистрационным картам;
- поиск по полным текстам.

Первый вид поиска соответствует случаю, когда пользователь что-то знает о документе, например, название, автора, дату выпуска и т.д. Самый простой случай, когда пользователь знает все. Тогда просто анализируется таблица РК, из нее отбирается нужная регистрационная карта, из которой отбирается указатели на страницы полного текста документа. Далее эти страницы выбираются из таблицы ПТ.

Несколько сложнее поиск в случае, когда пользователь знает только часть атрибутов регистрационной карты, например, только одно название или только словосочетание из названия. В этом случае предварительно анализируется словарь и инверсная таблица РК, после чего отыскивается сама РК.

Полнотекстовый поиск соответствует ситуации, когда пользователь ничего не знает о документе и может указать только ключевые слова для него. В этом случае прежде всего используется инверсная таблица ПТ, из которой отыскивается список страниц, содержащих эти слова. Если такой список оказывается очень велик, может быть использован словарь ПТ, позволяющий сократить его в соответствии с частотой появления слов.

Соответствие терминологии, принятой в реляционной модели БД и документной модели БД приведено в таблице 1.3.

Табл. 1.3. Соответствие терминов

Реляционная модель БД	Документная модель БД
Схема данных	База данных
Таблица	Коллекция документов
Строка	Документ
Идентификатор строки	ID документа

Документные базы данных обеспечивают разнообразные функциональные возможности запросов.

Одним из преимуществ документных баз данных по сравнению с хранилищами типа «ключ-значение» является то, что можно послать запрос к содержанию документа, не извлекая весь документ по его ключу, чтобы просмотреть его. Это свойство делает такие базы данных похожими на модель запроса реляционных БД.

Рассмотрим некоторые запросы к базе данных MongoDB, в которой можно хранить документы.

Допустим, мы хотим вернуть все документы из коллекции заказов (все строки из таблицы заказов). На языке SQL этот запрос выглядит так:

```
SELECT *
FROM Заказы;
```

Эквивалентный запрос в оболочке базы Mongo выглядит следующим образом:

```
db. Заказы. find ()
```

Выбор заказов для конкретного идентификатора, равного 883c2c5Ь4e5Ь, можно записать так:

```
SELECT *
FROM Заказы
WHERE ID_покупателя = "883c2c5Ь4e5Ь";
```

Эквивалентный запрос в базе Mongo на получение всех заказов для конкретного идентификатора выглядит следующим образом:

```
db.order.find({" ID_покупателя":"883c2c5b4e5b"})
```

Аналогично выбрать записи orderld и orderDate для заданного клиента на языке SQL можно записать так:

```
SELECT orderId,orderDate
FROM order
WHERE customerId = "883c2c5Ь4e5Ь"
```

Эквивалент этого запроса в базе Mongo можно записать следующим образом:

```
db.Заказы.find({ID_покупателя:"883c2c5b4e5b"}, {ID_заказа:1, Дата_заказа:1})
```

Аналогично можно формировать запросы для подсчета, суммирования и выполнения других операций.

Преимущества документных БД в сравнении реляционными БД:

1. В сравнении с реляционными базами данных лучшая производительность при индексировании больших объемов данных и большим количестве запросов на чтение.
2. Легче масштабируются в сравнении с SQL решениями.
3. Децентрализированы – работа на скластере.
4. Легко менять «схему» данных: не нужно выполнять никаких операций обновления для добавления новых полей.
5. Хранение неструктурированных данных.
6. Единое место хранения всей информации об объекте, что требует меньше операций вида «join».
7. Простой интерфейс общения с БД (ключ → значение, нет SQL).

Недостатки документных БД:

1. Отсутствие транзакционной логики и контроля целостности в большинстве реализаций: необходимо реализовывать ее в логике приложения.
2. Для обработки данных необходимо использование дополнительного языка программирования.

Рассмотрим примеры, где документные БД подходят лучше всего:

Регистрация событий.

Приложения по-разному регистрируют события; на предприятиях существует множество разных приложений, желающих регистрировать события. Документные базы данных могут хранить все эти типы событий и действовать как центральное хранилище событий. Это особенно важно в ситуациях, когда тип данных, собираемых событиями, постоянно изменяется.

Системы управления информационным наполнением, блог-платформы.

Поскольку документные базы данных не имеют предопределенной схемы и обычно понимают JSON-документы, они хорошо работают в системах управления информационным наполнением или в приложениях по публикации веб-сайтов, управляющих комментариями пользователей, их регистрацией, профилями, а также представлением документов в веб.

Веб-аналитика и аналитика в реальном времени.

Документные базы данных могут хранить данные, необходимые для анализа в реальном времени; поскольку части документов можно обновлять, можно очень легко хранить представления страниц или информацию об отдельных посетителях, а также добавлять новые показатели эффективности без изменения схемы.

Приложения для электронной коммерции.

Приложения для электронной коммерции часто должны иметь гибкую схему товаров и заказов, а также возможность изменять свои модели данных без дорогостоящего перепроектирования кода базы данных или обновления структуры базы.

Документные БД не рекомендуется использовать в следующих задачах.

Сложные транзакции, охватывающие разные операции.

Если есть необходимость выполнять атомарные операции с несколькими документами, то документные базы данных для этого, как правило, не подходят.

Запросы к изменяющейся агрегатной структуре.

Гибкая схема означает, что база данных не накладывает на схему никаких ограничений. Данные сохраняются в виде сущностей приложения. Если возникает необходимость в специальном запросе к этим сущностям, то запросы можно изменять. Поскольку данные сохраняются как агрегат, то при постоянном изменении структуры агрегата необходимо сохранять его с наименьшим уровнем детализации, т.е. фактически нормализовать данные. В таком сценарии документная база данных может оказаться неработоспособной.

Изучение особенностей документных баз данных позволяет сделать следующие выводы:

1. Единицей хранения в документных БД является документ, основная часть которого – неструктурированный текст.
2. Документальная БД включает в себя как минимум три области хранения данных: файл словаря, инверсный список, текстовый файл.
3. Поиск в документных БД может быть реализован по метаданным (регистрационным картам) или ключевым словам непосредственно в контенте документа.
4. Схема данных отсутствует. Не нужно выполнять никаких операций обновления для добавления новых полей.
5. В системе управления документными БД отсутствует транзакционная логика и контроль целостности.
6. Для обработки данных необходимо использование дополнительного языка программирования.

1.5. Базы данных «семейство столбцов»

Семейство столбцов (столбчатая БД) – это коллекция строк, содержащая множество столбцов, ассоциированных с ключом строки. Семейства столбцов группируют взаимосвязанные данные, доступ к которым обеспечивается как к единому целому.

Основной единицей хранения в базе данных является столбец, состоящий из пары «имя-значение», в которой имя играет роль ключа. Каждая из пар «ключ-значение» может храниться с меткого времени, которая используется для того, чтобы задавать срок действия данных, разрешать конфликты записи, обрабатывать устаревшие данные и выполнять другие функции.

Модель семейства столбцов можно представить, как двухуровневую агрегатную структуру. Как и в хранилищах типа «ключ-значение», главный ключ – это идентификатор строки, отмечая нужный в данный момент агрегат.

Отличительной особенностью «семейства столбцов» является то, что строка-агрегат сама состоит из ассоциативного массива более детализированных значений. Эти значения второго уровня называются столбцами (см. рис. 1.13).

Помимо доступа к строкам как к единому целому, операции также допускают извлечение конкретного столбца, так что, для того чтобы получить имя клиента в БД на рисунке 1.13 можно написать команду наподобие get ('1234', 'name').

Столбцы организуются в семейства. Каждый столбец является частью одного семейства столбцов и единица доступа. Данные в конкретном семействе столбцов обычно доступны одновременно.

Итак, данные в столбчатой базе данных структурированы следующим образом:

– Ориентация по строкам: каждая строка – это агрегат (например, клиент с идентификатором 1234), а семейства столбцов содержат фрагменты данных (профиль, история заказов) в этом агрегате.
– Ориентация по столбцам: каждое семейство столбцов определяет тип записи (например, профили клиентов), причем каждой записи соответствуют строки. В таком случае строку можно интерпретировать как объединение записей из всех семейств столбцов.

Последний аспект отражает «столбцовую» природу баз данных типа «семейство столбцов». Базы данных этого типа имеют двумерный характер.

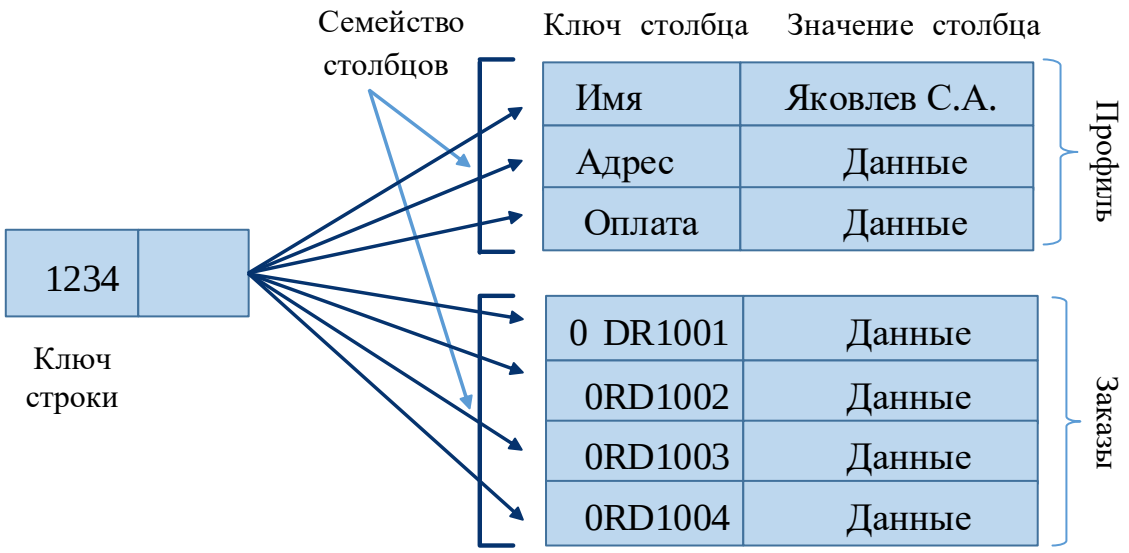

Рисунок 1.13 – Столбчатая БД

Представлять семейства столбцов в виде таблиц неправильно, т.к. в этой БД можно добавлять любой столбец в любую строку, а строки могут иметь самые разные ключи.

Новые столбцы добавляются в строки при обычном доступе к базе данных.

Определение нового семейства столбцов происходит намного реже и может вызвать остановку работы базы данных.

Поскольку столбцы можно добавлять свободно, список элементов можно легко моделировать, сделав каждый элемент отдельным столбцом.

Строка семейства столбцов – это агрегат. Строки бывают широкими и «худыми».

«Худые» строки содержат несколько столбцов, причем одни и те же столбцы используются в разных строках. В данном случае семейство столбцов определяет тип записи, каждая строка является записью, а каждый столбец – полем.

Широкая строка содержит много разных столбцов (возможно, тысячи). Широкое семейство столбцов моделирует список, в котором каждый столбец представляет собой элемент в этом списке. Широкие семейства столбцов могут определять определенный порядок следования своих столбцов.

Каждая из пар «ключ-значение» всегда хранится с меткого времени, которая используется для того, чтобы задавать срок действия данных, разрешать конфликты записи, обрабатывать устаревшие данные и выполнять другие функции. Если данные столбца больше не используются, то это место можно восстановить позднее на этапе уплотнения. Рассмотрим пример:

{

Имя: "Полное имя",
Значение: "Яковлев Сергей",
Метка: 12345667890
}

Здесь столбец «Имя» содержит ключ строки «Полное имя» и значение «Яковлев Сергей», а также связанную с ними метку времени «12345667890».

Строка – это коллекция столбцов, ассоциированных с ключом; коллекция таких строк образует семейство столбцов. Если семейство столбцов состоит из обычных столбцов, оно называется стандартным.

Каждое семейство столбцов можно сравнить с контейнером строк в таблице реляционной БД, в которой ключ идентифицирует строку, а строка состоит из множества столбцов. Отличие заключается в том, что разные строки не обязаны содержать одинаковые столбцы, причем столбцы могут добавляться в любую строку в любое время и добавлять их в остальные строки не обязательно, например:

```
//Семейство столбцов
{//row
  "Яковлев Сергей": {
      Имя: "Сергей",
      Фамилия: "Яковлев",
      Последний визит: "2019/05/12"
  }
//Строка
 "Татарникова Татьяна": {
     Имя: "Татьяна",
     Фамилия: "Татарникова",
     Город: "Санкт-Петербург"
  }
}
```

В этом примере строки «Яковлев Сергей» и «Татарникова Татьяна» имеют разные столбцы; обе эти строки являются частью семейства столбцов.

Соответствие терминологии, принятой в реляционной модели БД и документной модели БД приведено в таблице 1.4.

Табл. 1.4. Соответствие терминов

Реляционная модель БД	Столбчатая БД
Атрибут	Столбец
Строка	Коллекция столбцов
Идентификатор строки	Ключ строки
Таблица	Агрегат (семейство столбцов)

Рассмотрим примеры, где БД типа «семейство столбцов» подходят лучше всего:

Регистрация событий.

Семейства столбцов, способные хранить любые структуры данных, приспособлены для хранения информации о событиях, например, состояние приложения или ошибки, обнаруженные приложением.

Системы управления информационным: наполнением, блог-платформы.

С помощью семейств столбцов можно хранить записи блогов с ключевыми словами, категориями, ссылками и обратными ссылками в разных столбцах. Комментарии можно хранить либо в той же строке, либо переместить в другое пространство ключей; аналогично пользователей блога и актуальные блоги можно поместить в разные семейства столбцов.

Счетчики.

В веб-приложениях часто возникает необходимость подсчета и классификации посетителей, чтобы вычислить аналитические показатели веб-страницы. После создания семейства столбцов каждому пользователю веб-приложения можно выделить произвольное количество столбцов для посещенных им веб-страниц.

Срок действия.

Иногда возникает необходимость создать демоверсии для пользователей или в определенное время размещать на веб-сайте рекламные объявления. Для этого можно использовать столбцы с ограниченным сроком действия: то есть создавать столбцы, которые автоматически удаляются по истечении определенного периода времени. Это время называется TTL (Time To Live – время существования) и задается в секундах. По истечении периода TTL столбец удаляется; если столбец больше не существует, запрос можно отменить, а рекламное объявление снять.

Существуют проблемы, которые семейство столбцов решает неэффективно. Например, это относится к системам, использующим для выполнения операций чтения и записи транзакции. Если необходимо, чтобы база данных агрегировала данные с помощью запросов (например, SUM или AVG), это следует делать на клиентской стороне с помощью данных, извлеченных из всех строк.

Столбчатые БД не стоит использовать для создания ранних прототипов или первичных промышленных систем: на ранних стадиях еще не известно, как изменится шаблон запросов, а изменяя шаблоны запросов, мы вынуждены изменять проект семейства столбцов.

Изучение особенностей баз данных типа «семейство столбцов» позволяет сделать следующие выводы:

1. Коллекция столбцов – это строка (агрегат).
2. Коллекция строк образует семейство столбцов.
3. Единицей хранения является столбец.
4. Столбцы могут добавляться в любую строку в любое время.
5. БД типа «семейство столбцов» в основном используются для регистрации событий и счета.
6. В столбчатых БД отсутствуют средства реализации итоговых функций и транзакций.

1.6. Графовые БД

Основу графовой модели БД образуют маленькие записи со сложными связями. В такой БД граф – это не диаграмма, а структура данных с узлами, соединенными ребрами.

На рисунке 1.14 показана веб-информация с очень маленькими узлами и многочисленными связями между ними. Работая с этой структурой, мы можем

задавать вопросы вроде «найти книгу в категории «Базы данных», написанную кем-то, чей друг мне нравится.

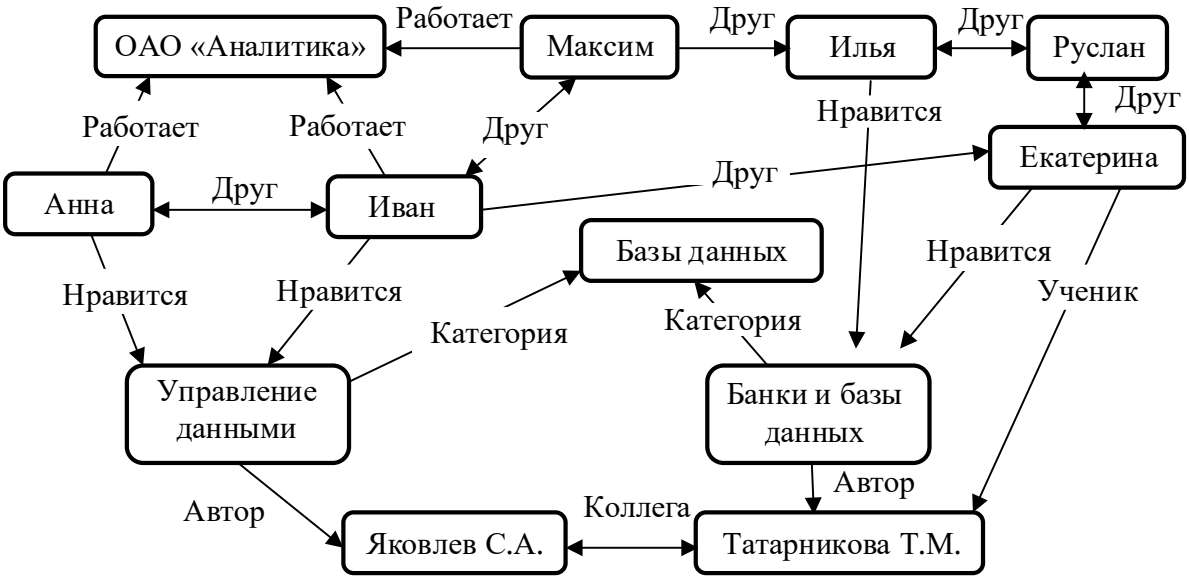

Рисунок 1.14 – Пример графовой БД

Графовые базы данных специально предназначены для хранения такой информации, но в более крупном масштабе, чем можно показать на диаграмме. Они идеально подходят для хранения любых данных, связанных со сложными отношениями, например, социальных сетей, товарных предпочтений или правил приема на работу.

Фундаментальная модель данных графовых баз очень простая: узлы – это сущности, соединенные ребрами (связями). Узлы имеют свойства. Ребра могут иметь свойства и направление.

Организация графа позволяет один раз записать данные, а затем интерпретировать их разными способами в соответствии со связями.

Как только построен граф узлов и ребер, база данных позволит послать к ней запрос. В этом проявляется важное различие между графовыми и реляционными базами данных.

Несмотря на то что реляционные базы данных могут реализовывать связи с помощью внешних ключей, операции соединения требуют навигации, которая может оказаться затратной. Следовательно, в моделях данных с большим количеством связей производительность упадет.

В графических базах данных обход узлов требует очень небольших затрат. В основном это объясняется тем, что графовые базы данных переносят большую часть работы, связанной с навигацией по связям с момента запроса на момент вставки. Это естественно оправдывает себя в ситуациях, когда производительность запроса важнее скорости вставки. Большую часть времени вы ищете данные, перемещаясь по ребрам сети с запросами вроде таких:

- «найти людей с именем Анна»;
- «найти книгу в категории «Базы данных»
- «найти автора(ов) книги «Управление данными»»;
- «найти, чем интересуется Иван» и т.д.

Однако для организации поиска необходима отправная точка, поэтому некоторые узлы могут быть индексированы атрибутом, например, идентификатором. Таким образом, можно начать с поиска идентификатора (например, найти людей с именами Анна и Барбара), а затем начать перемещение по ребрам.

Как следует из рассмотренных свойств, графовые базы предназначены для задач, в которых большую часть времени выполняется поиск – происходит перемещение по связям.

Акцент на связях сильно отличает графовые базы данных от агрегатно- ориентированных. Это отличие проявляется в том, что, во-первых, графовые базы данных чаще работают на одном сервере, а не распределены по кластерам, а во-вторых, в графовых БД реализован механизм транзакций, обеспечивающий согласованность данных.

Единственное, что связывает их с агрегатно-ориентированными базами данных – это отрицание реляционной модели.

Отношения (связи) между узлами создаются в двух направлениях. Рассмотрим графовую БД, изображенную на рисунке 1.15. Например, узел Анна работает в ОАО «Аналитика», а узел Екатерина нет.

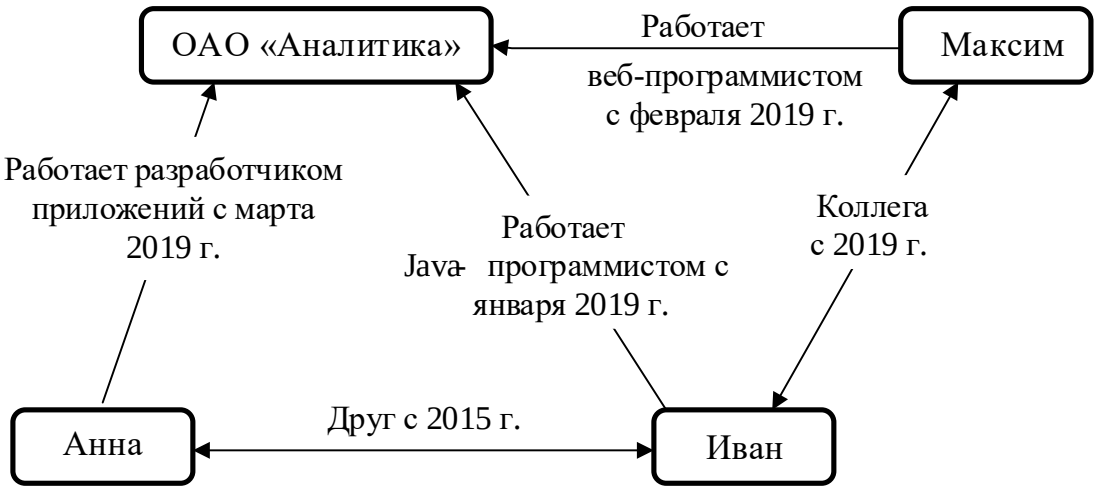

Рисунок 1.14 – Графовая БД

Направленность отношения позволяет проектировать сложные предметно-ориентированные модели. Известные входящие и исходящие отношения можно обходить в обоих направлениях.

Отношения являются полноправными элементами графовых баз данных. Собственно, ценность графовых баз данных в основном обусловлена отношениями. Отношения имеют не только тип, начальный и конечный узел, но и свои собственные свойства. Используя эти свойства, в отношение можно внести информацию, например, такую: когда узлы стали «друзьями», каково расстояние между узлами и что между ними общего. Эти свойства отношений можно использовать при создании запроса к графу.

Так как мощь графовых баз данных в основном обеспечивается отношениями и их свойствами, требуется серьезная аналитическая и конструкторская работа по моделированию отношений между объектами предметной области. Добавление новых типов узлов в графовую БД выполняется просто. Изменение существующих узлов и связей между ними эквивалентно осуществлению миграции данных, потому что эти изменения необходимо вносить в каждом узле и в каждом отношении между существующими данными.

Графовые БД поддерживают транзакции. Прежде чем изменить какой-нибудь узел или добавить какое-то отношение к существующим узлам, необходимо начать транзакцию, в которую упаковываются операции изменения. Исключение составляет только операция чтения. Чтение можно выполнять без создания транзакций.

Ниже приведен код транзакции, которая применяется к БД, граф которой приведен на рисунке 1.14, и создает в ней узел и задает его свойства. Здесь транзакция начата функцией success и закончена функцией finish:

```
Transaction.success = database.beginTx();
try {
    Node друг = GraphDB.createNode();
    друг.setProperty("name", “Илья");
    Максим.createRelationshipTo(Илья, друг);
    Иван.createRelationshipTo(Илья, друг);
    transaction.success();
} finally {
    transaction.finish();
```

На рисунке 1.15 приведен граф БД после выполнения транзакции на добавление нового узла и его свойств.

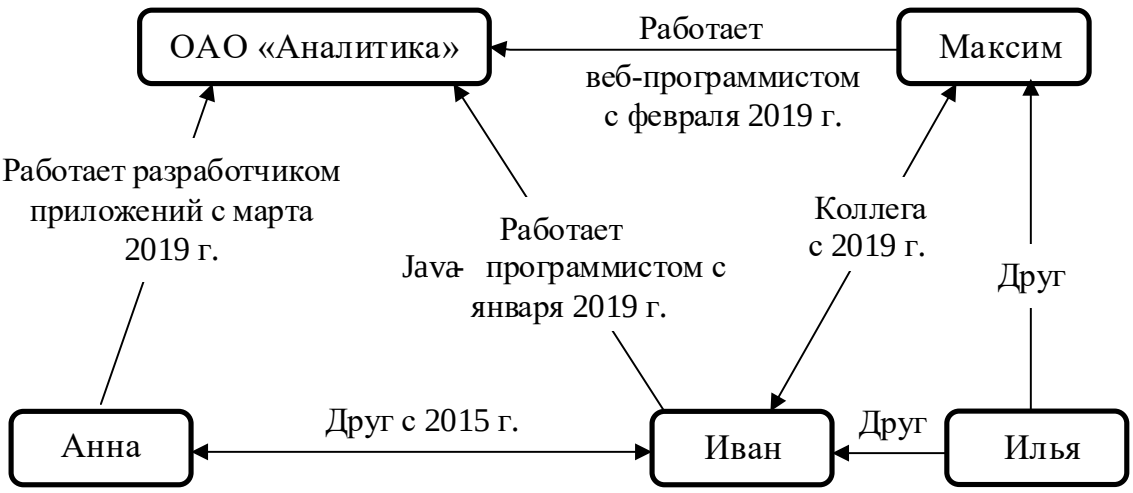

Рисунок 1.15 – Графовая БД после добавления нового узла

Рассмотрим возможности запросов, которые применяются в графовых БД.

Поскольку свойства узла индексируются и свойства отношений также индексируются, то узел или ребро можно найти по значению индекса. Индексы узлов задаются либо в момент их добавления в базу данных, либо при их последующем обходе.

Если ищем узел Анна в БД на рисунке 1.15, то можем запросить индекс для свойства name, имеющего значение Анна. Имея этот узел, можем выяснить все его отношения, в том числе входящие и исходящие.

При запросе свойств связей можно применять фильтры направлений.

Настоящая мощь графовых БД проявляется в ситуациях, когда необходимо обойти граф на любой глубине и указать начальную точку обхода. Это особенно полезно при попытках найти узлы, связанные с начальной точкой, на более чем одном уровне глубины.

Одно из преимуществ графовых БД – это разнообразие возможностей поиска путей между двумя узлами: можно определить, есть ли несколько путей, найти все пути или кратчайший путь. Эта функциональная возможность используется в социальных сетях для демонстрации отношений между двумя узлами.

Соответствие терминологии, принятой в реляционной модели БД и графовой модели БД приведено в таблице 1.5.

Табл. 1.5. Соответствие терминов

Реляционная модель БД	Столбчатая БД
Атрибут	Узел
Строка	Подграф
Идентификатор строки	Индекс узла
Таблица	Граф

Рассмотрим примеры, где БД типа «граф» подходят лучше всего:

Связанные данные.

Графовые данные можно развернуть и очень эффективно использовать в социальных сетях. Вообще любая предметная область с богатыми взаимными связями подходит для описания с помощью графа. Если в одной и той же базе данных существуют отношения между сущностями из разных предметных областей (например, социальные, географические и коммерческие связи), можно повысить их информативность, предусмотрев возможность обхода графа с пересечением границ предметных областей.

Маршрутизация, диспетчеризация и геолокационные сервисы.

Каждое место назначения или адрес можно представить в виде узла, а все узлы, в которые необходимо выполнить доставку, можно моделировать с помощью графа. Отношения между узлами могут иметь отношение, описывающее

расстояние. Это позволяет обеспечить эффективную доставку товаров. Свойства расстояния и адреса можно использовать в графах, описывающих предпочтения пользователей, так что приложение может выдавать рекомендации о хороших ресторанах или местах для развлечений, расположенных поблизости. Можно также создать узлы для понравившихся точек, например, ресторана, и уведомить пользователей, что они находятся поблизости, используя геолокационную службу.

Справочные базы данных.

После того как в системе созданы узлы и отношения, их можно использовать для выдачи рекомендаций типа «ваши друзья также купили этот товар» или «при заказе этого товара обычно также заказывают следующие товары». У таких справочных баз данных есть один интересный побочный эффект – при увеличении объема баз данных количество узлов и отношений, доступных для выдачи рекомендаций, быстро увеличивается.

В некоторых ситуациях графовые базы данных могут оказаться неприемлемыми:

– при необходимости обновлять все или часть сущностей, например, при выработке аналитического решения, в котором свойства всех сущностей могут быть изменены. Изменение свойства во всех узлах является сложной труднореализуемой операцией;
– при выполнении глобальных операций над графом, то есть тех, что затрагивают весь граф.

Изучение особенностей баз данных типа «граф» позволяет сделать следующие выводы:

1. Графовая модель БД – это информационные объекты и отношения между ними, представленные в виде графа.
2. Графовая модель БД не является агрегатно-ориентированной.
3. Единицей хранения в графовых БД является узел (объект) с приписанными ему свойствами
4. Основная операция, выполняемая в графовых БД – поиск информации.
5. Для реализации поиска в графовых БД необходимо выполнить индексацию узлов и отношений между ними.
6. Изменения в структуре графовой модели БД выполняются с помощью транзакций.
7. Возможность поиска путей между двумя узлами.