Масштабируемость баз данных и модели данных: изучение различных подходов в реляционных, NoSQL и OLAP-системах

Масштабируемость баз данных включает в себя фундаментальные компромиссы между согласованностью, производительностью и сложностью различных систем данных. Для эффективного масштабирования необходимо понимать, как базы данных распределяют работу, их базовые предположения о моделях доступа к данным и связанные с этим архитектурные компромиссы.

В данном анализе представлен обзор подходов к масштабированию баз данных с акцентом на практические критерии принятия решений для производственных систем.

Масштабируемость баз данных и модели данных: систематические критерии выбора

По мере того как количество пользователей приложений растет от сотен до миллионов, архитектура баз данных становится критически важной для производительности системы. Успех зависит от понимания того, как масштабируются различные базы данных, и от выбора подходящих моделей данных и подходов к масштабированию для конкретных моделей рабочей нагрузки.

Понимание масштабируемости баз данных

Масштабируемость — это не только обработка большего объема данных, но и поддержание производительности, надежности и разумных затрат по мере роста системы. Задача состоит в том, чтобы сбалансировать несколько конкурирующих факторов:

Модели чтения и записи

некоторые приложения предполагают интенсивное чтение (например, сайты с контентом), другие — интенсивную запись (например, сбор данных IoT). Каждая модель требует разных стратегий масштабирования.

Направление масштабирования

вы можете масштабировать вверх, покупая более мощные серверы (вертикальное масштабирование), или масштабировать вшир, добавляя больше серверов (горизонтальное масштабирование). Масштабирование вверх проще, но имеет жесткие ограничения; масштабирование вшир сложнее, но позволяет расти практически без ограничений.

Компромисс между репликацией и шардингом

репликация копирует весь набор данных на несколько серверов, увеличивая пропускную способность чтения и обеспечивая резервное копирование. Шардинг разделяет данные между серверами, увеличивая емкость записи, но усложняя запросы, охватывающие несколько фрагментов.

- Шардинг -> помогает масштабировать запись и хранение
- Репликация -> помогает масштабировать чтение
- Репликация -> помогает обеспечить высокую доступность и отказоустойчивость

Согласованность и производительность: Сильная согласованность (все серверы сразу видят одни и те же данные) замедляет работу. Конечная согласованность (серверы синхронизируются со временем) быстрее, но может отображать устаревшие данные.

Реляционные базы данных: адаптированная основа

Реляционные базы данных предлагают прекрасную предсказуемость. Они заставляют заранее четко продумать структуру данных — определить таблицы, настроить внешние ключи, нормализовать отношения — и затем SQL делает именно то, что от него ожидают.

Реляционная модель в основном касается структуры и отношений. Данные хранятся в таблицах с жесткой схемой, где каждая строка следует одной и той же структуре столбцов. При соединении связанных данных, таких как пользователи и их заказы, внешние ключи и соединения обеспечивают явные отношения. Иногда это многословно, но явно. Не нужно гадать, как выглядят данные или как они связаны.

SQL остается одним из самых мощных языков запросов, когда-либо созданных. Возможность выражать сложные аналитические вопросы в нескольких строках декларативного кода, не беспокоясь о том, как движок базы данных оптимизирует выполнение, просто поразительна. Младшие разработчики могут писать сложные отчеты с подзапросами и окнами, которые в других системах заняли бы сотни строк процедурного кода.

Несмотря на прогнозы о их исчезновении, реляционные базы данных заметно эволюционировали. Современные PostgreSQL и MySQL обрабатывают документы JSON, полнотекстовый поиск и геопространственные запросы — функции, которые когда-то были доступны только системам NoSQL.

PostgreSQL/MySQL: ограничения масштабируемости записи

PostgreSQL является примером того, как традиционные базы данных адаптировались к современным требованиям. Его архитектура с одним мастером означает, что все записи проходят через один сервер, но он отлично справляется с масштабированием чтения благодаря потоковой репликации. Вы можете запустить несколько реплик для чтения, которые обслуживают запросы, пока главный сервер обрабатывает записи.

Этот подход отлично подходит для рабочих нагрузок с интенсивным чтением. Новостной сайт может использовать один главный сервер для публикации контента и пять реплик для чтения, чтобы обслуживать миллионы читателей. Каждая реплика может обрабатывать тысячи запросов в секунду с временем отклика 10–50 мс. Облачные сервисы, такие как AWS Aurora, идут еще дальше, отделяя хранение от вычислений. Aurora утверждает, что пропускная способность в 3 раза выше, чем у стандартного PostgreSQL, и может автоматически масштабировать хранилище, сохраняя время запроса менее 100 мс.

> Но все же в большинстве конфигураций PostgreSQL по-прежнему использует **архитектуру с одним мастером** для записи — все операции записи должны проходить через один основной экземпляр: он обрабатывает все операции INSERT, UPDATE, DELETE и DDL.

Еще одним недостатком этой конфигурации с несколькими репликами для чтения является необходимость ослабления свойств ACID. Я имею в виду, что может быть нецелесообразно заставлять БД синхронно реплицировать все записи в реплики перед возвратом результата пользователю. Ведь если одна синхронная реплика для чтения выходит из строя или работает медленно, записи должны приостанавливаться до устранения проблемы. PostgreSQL предлагает различные средства для управления этим поведением на уровне транзакций или на глобальном уровне. Альтернативным решением является разработка собственных стратегий для обеспечения требований к согласованности (например, обеспечение того, чтобы чтение измененных строк всегда проходило через мастер, чтобы гарантировать повторяемость чтения).

Когда необходимо масштабировать записи, PostgreSQL требует большего творческого подхода. Вы можете разбивать большие таблицы по дате или ID пользователя или использовать расширения, такие как Citus (https://www.citusdata.com/), которые распределяют таблицы по нескольким узлам, сохраняя совместимость с SQL. Команды сообщают, что с правильно настроенными кластерами Citus они обрабатывают более 30 000 вставок в секунду.

Для масштабирования в огромных масштабах такие компании, как Facebook, создали собственные системы шардинга MySQL. Они разбивают пользовательские данные на тысячи экземпляров MySQL, а логика приложения направляет запросы на правильный шард на основе идентификатора пользователя.

Другой подход — репликация с несколькими лидерами, при которой несколько реплик могут принимать записи. Он изначально поддерживается некоторыми базами данных: MySql (функция Group Replication), Oracle, SQL Server и YugabyteDB или с помощью дополнения (например, в Redis Enterprise, EDB Postgres Distributed и pglogical). Однако репликация с несколькими лидерами часто считается опасной территорией для RDBMS, поскольку часто возникают тонкие ошибки конфигурации и неожиданные взаимодействия с другими функциями базы данных: например, могут возникнуть проблемы с ключами с автоинкрементом, триггерами и ограничениями целостности. Решение конфликтов также может стать кошмаром (например, необходимо определить, как разрешать конфликты, когда две реплики записи принимают записи в пересекающийся набор значений).

Документные базы данных: созданы для распределения

Документные базы данных идеально подходят для многих современных приложений. Вместо того чтобы навязывать объектно-ориентированное мышление в виде строк и столбцов, они позволяют хранить данные в естественном виде — в виде документов с вложенными структурами. Переход от реляционного хранения к хранению документов часто воспринимается как освобождение. Хранение профиля пользователя со встроенной информацией об адресе, настройках и журналами действий в виде одного документа устраняет необходимость разделять объекты на несколько нормализованных таблиц или использовать сложные соединения для их повторной сборки. Объекты приложения напрямую сопоставляются с документами базы данных.

Гибкость — это и благо, и проклятие. Каждый документ в коллекции может иметь совершенно разные поля — один пользователь может иметь адрес доставки, а другой — нет, один продукт может иметь подробные характеристики, а другой — быть простой единицей товара. Эта гибкость схемы ускоряет разработку и прекрасно адаптируется к меняющимся требованиям. Но это также означает потерю некоторых средств безопасности, которые обеспечивают жесткие схемы.

Языки запросов в документах баз данных кажутся более естественными для разработчиков, имеющих опыт программирования. Синтаксис запросов MongoDB напоминает объекты JavaScript. Вы можете напрямую запрашивать вложенные поля, фильтровать массивы и запускать конвейеры агрегации, которые больше похожи на функциональное программирование, чем на декларативный подход SQL.

MongoDB: гибкость и масштабируемость

MongoDB с самого начала разрабатывался с учетом горизонтального масштабирования. В отличие от RDBMS с одним мастером, он может масштабировать записи по горизонтали с помощью шардинга.

В кластере MongoDB с шардингом каждый шард имеет свой собственный первичный сервер, который может принимать записи одновременно, поэтому записи распределяются между несколькими первичными серверами на основе ключа шарда. В то же время один узел может быть первичным для одного шарда и одновременно вторичным (не первичным) для других шардов. Это позволяет пропускной способности записи масштабироваться линейно с количеством шардов, а не создавать узкие места через один главный узел.

Его система шардинга автоматически распределяет коллекции по нескольким серверам на основе выбранного вами ключа шарда. Каждый шард сам по себе является набором реплик, обеспечивая как масштабируемость, так и доступность.

01

Маршрутизация запросов

Все происходит на уровне маршрутизатора mongos. Когда ваше приложение запрашивает сообщения пользователя, mongos проверяет ключ фрагмента и направляет запрос в правильный фрагмент.

02

Обработка данных

Уровень маршрутизации mongos не имеет состояния и горизонтально масштабируем (вы можете запустить несколько экземпляров mongos), в отличие от единственного мастера, который фактически обрабатывает и хранит данные — mongos просто направляет запросы в соответствующие первичные фрагменты, которые выполняют реальную работу.

03

Координация запросов

Для запросов, охватывающих несколько шардов, он координирует операцию распределения и сбора по соответствующим шардам.

Эта архитектура отлично подходит для приложений с гибкими схемами и высокой нагрузкой на запись. Социальная сеть может разбивать сообщения пользователей по идентификаторам, позволяя тысячам одновременных пользователей публиковать сообщения, не конкурируя за одни и те же ресурсы базы данных. Хорошо настроенные кластеры MongoDB обычно обрабатывают тысячи записей в секунду с задержкой 10–100 мс.

Сложность заключается в выборе правильного ключа фрагмента. Неудачный выбор может привести к появлению «горячих точек», где один фрагмент получает большую часть трафика, а остальные простаивают. Это требует глубокого понимания ваших шаблонов запросов.

Широкостолбцовые хранилища: линейная масштабируемость

Название «широкостолбцовый» — один из самых запутанных терминов в базах данных. Несмотря на то, что название предполагает обратное, он не имеет ничего общего со столбцовым хранением. Это означает, что каждая строка может иметь свой набор столбцов, что делает таблицы «широкими» в том смысле, что они могут содержать очень разные структуры данных в одной логической таблице.

Такой подход удобен для хранения **полуструктурированных данных**, где заранее определить схему невозможно или нецелесообразно. Широкостолбцовые хранилища находятся где-то между реляционными структурами и документными БД без жесткой схемы.
Наборы столбцов группируются в **семейства** (column families), и это позволяет разделять **данные внутри строки** на логические группы — например, *«горячие»* (часто читаемые) и *«холодные»* (редко читаемые).
Это **не столбцовое хранение**, но **даёт похожий эффект**: ты можешь прочитать только нужную часть данных, не загружая всё остальное.

Языки запросов здесь прагматично ограничены. CQL Cassandra внешне похож на SQL, но не имеет join'ов и сложных запросов, которые ожидают разработчики. Все вращается вокруг ключа раздела — запросы должны быть построены с учетом распределения данных, а не наоборот. Это кажется отступлением от SQL, но как только к этому привыкаешь, прирост производительности становится заметным.

> Терминология, связанная с «широкостолбцовыми» хранилищами, действительно сбивает с толку, потому что, несмотря на схожесть названий, она не имеет ничего общего с столбцовым хранением. Широкостолбцовые хранилища, такие как Cassandra, относятся к гибкой модели данных, в которой каждая строка может иметь много разных столбцов (что делает таблицы «широкими»), а разные строки могут иметь совершенно разные наборы столбцов, в отличие от хранилищ ключей-значений, где каждый ключ сопоставляется ровно одному блоку значений.

Речь идет исключительно о логической структуре данных — хранилища с широкими столбцами на самом деле используют внутреннее хранение на основе строк, сохраняя все столбцы для каждой строки вместе на диске, что противоположно столбцовым системам хранения, таким как ClickHouse, которые физически хранят все значения для каждого столбца вместе, чтобы оптимизировать аналитические запросы. «Широкий» в «широкостолбцовом» не имеет ничего общего с тем, как данные хранятся на диске, а имеет отношение к гибкой схеме, которая может вместить много столбцов в каждой строке, что делает это одно из самых запутанных названий в терминологии баз данных.

Cassandra: чемпион по записи

Cassandra использует радикально другой подход — все узлы равны. Нет главного сервера; вместо этого данные распределяются по узлам с помощью согласованного хеширования. Любой узел может принять запрос и выступить в роли координатора, используя согласованное хеширование для определения узлов, которые должны обработать данные, а затем либо обработать их локально (если он владеет данными), либо переслать их соответствующим узлам — нет отдельного уровня маршрутизации, как в MongoDB mongos.

При записи данных Cassandra может записывать на несколько узлов одновременно, что делает ее исключительно эффективной при обработке рабочих нагрузок с интенсивной записью.

Эта одноранговая архитектура означает, что добавление узлов напрямую увеличивает как пропускную способность чтения, так и пропускную способность записи. Netflix известен тем, что использует кластеры Cassandra, которые обрабатывают более 200 000 записей в секунду со средней задержкой около 1–2 мс.

200K	1-2
Записей в секунду	Миллисекунды
Netflix обрабатывает в кластерах Cassandra	Средняя задержка записи

Компромиссом является сложность. Cassandra предоставляет вам тонкий контроль над согласованностью — вы можете выбрать запись только на один узел для максимальной скорости или потребовать подтверждения записи большинством узлов для более высокой согласованности. Эта гибкость очень мощная, но требует тщательной настройки.

Cassandra отлично подходит для таких сценариев, как телеметрия IoT, где миллионы датчиков непрерывно передают данные. Возможность настраивать согласованность для каждого запроса позволяет приоритезировать скорость поступления данных, требуя при этом более высокой согласованности для критически важных чтений.

Хранилища ключей-значений: простота и скорость

Иногда лучшее решение — самое простое. Хранилища ключей-значений сводят базы данных к самому необходимому — у вас есть ключ, он указывает на значение, и все. Никаких схем, никаких отношений, никакого сложного планирования запросов. Только молниеносный поиск.

Современные хранилища ключей-значений, такие как Redis, вышли за рамки простого хранения строк. «Значение» может быть строкой, списком, набором, отсортированным набором, хэшем или даже более сложной структурой данных, такой как растровое изображение или HyperLogLogs. Это дает вам возможность выполнять базовые операции с данными — добавлять в список, добавлять в набор, увеличивать счетчик — при этом сохраняя простоту доступа на основе ключей и, что важно, обеспечивая атомарность многих операций.

Прелесть этой модели заключается в ее предсказуемости. Вы точно знаете, какую производительность получите, потому что важна только одна операция: поиск ключа. Базе данных неважно, является ли ваше значение простой строкой, JSON-блоком или двоичными данными — она просто хранит и извлекает все, что вы ей даете.

«Язык запросов» едва ли заслуживает такого названия. GET — ключ, PUT — значение, DELETE — когда все готово. Некоторые системы позволяют выполнять операции пакетами или атомарные обновления, но это все, на что они способны. Для тех, кто привык к SQL, это кажется почти примитивным, но именно эта простота делает эти системы такими быстрыми и надежными.

Рассмотрим две популярные БД ключ-значение: DynamoDB и Redis

DynamoDB: управляемая масштабируемость

Amazon DynamoDB представляет «бессерверный» подход к масштабированию. Вам не нужно управлять серверами или беспокоиться о шардинге — AWS обрабатывает все это за кулисами. Вы просто определяете свои требования к пропускной способности (или используете масштабирование по требованию), а DynamoDB автоматически разбивает ваши данные на разделы и настраивает емкость.

Этот управляемый подход обеспечивает стабильную задержку в пределах нескольких миллисекунд независимо от масштаба. Игровые компании используют DynamoDB для хранения данных сеансов игроков, где время отклика менее 5 мс критически важно для плавного игрового процесса. Сервис может обрабатывать тысячи запросов в секунду в часы пиковой нагрузки.

Миллисекунд

Критическое время отклика для игровых сессий

Стоимость... ну, это стоимость. DynamoDB может стать дорогостоящим при масштабировании, и вы будете привязаны к экосистеме AWS. Но для приложений, которым требуется предсказуемая производительность без операционных затрат, это очень привлекательное решение.

Redis: хранилище K/V в памяти

Redis работает в памяти, что делает его невероятно быстрым — время отклика составляет микросекунды. В кластерном режиме Redis разбивает данные по узлам с помощью хеш-слотов, что позволяет масштабировать как чтение, так и запись по горизонтали.

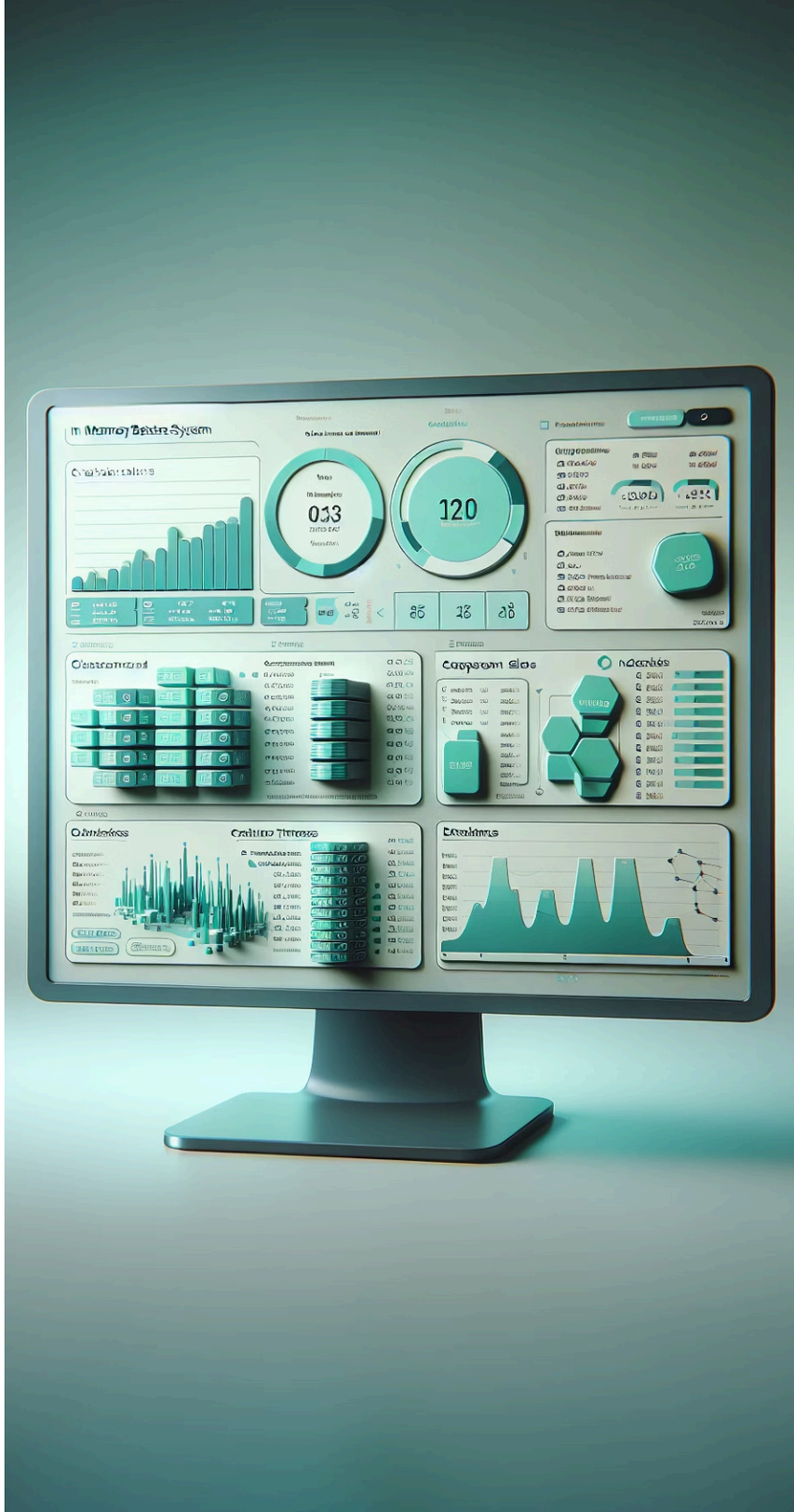
Классический случай использования — кэширование. Сайт электронной коммерции может кэшировать информацию о продуктах в Redis, обслуживая миллионы запросов страниц продуктов из памяти, в то время как основная база данных обрабатывает меньший объем заказов. Один экземпляр Redis может обрабатывать более миллиона операций в секунду.

1M+

Операций в секунду

Один экземпляр Redis может обрабатывать

Ограничение очевидно: все должно помещаться в памяти. Для больших наборов данных это быстро становится дорого. Но для «горячих» данных, требующих сверхнизкой задержки, Redis не имеет себе равных.



Поисковые системы: оптимизация для интенсивного чтения

Поисковые системы, такие как Elasticsearch, подходят к данным совершенно иначе, чем традиционные базы данных. Вместо того чтобы хранить документы и выяснять, как их запросить позже, они анализируют все заранее и создают инвертированные индексы — по сути, создают карту, связывающую каждое слово с каждым документом, в котором оно встречается.

Именно в этой предварительной обработке и заключается вся магия. Когда вы индексируете документ, Elasticsearch не просто сохраняет его — он разбивает текст, анализирует его и создает несколько индексов для разных типов запросов. Это требует больших вычислительных ресурсов на начальном этапе, но окупается, когда нужно просканировать миллионы документов и получить результаты за миллисекунды.

Язык запросов отражает эту ориентированность на поиск. Query DSL Elasticsearch невероятно мощный инструмент для поисковых операций — нечеткое сопоставление, оценка релевантности, сложная фильтрация, агрегирование по нескольким измерениям. Он может делать с текстовым поиском то, что было бы сложно или невозможно в SQL. Но попробуйте использовать его для транзакционных операций, и вы быстро поймете, что он не предназначен для такой работы.

Elasticsearch: аналитика в больших масштабах

Elasticsearch превосходно справляется с одной задачей: делает большие наборы данных доступными для поиска. Он автоматически разбивает индексы на фрагменты между узлами и может параллелизировать сложные поиски по всему кластеру. Это делает его идеальным решением для аналитики логов, где может потребоваться поиск по миллионам записей в режиме реального времени.

Архитектура приоритезирует производительность чтения. Когда вы индексируете документ, Elasticsearch анализирует текст, создает различные индексы и распределяет данные по фрагментам. Эта предварительная работа окупается, когда вам нужно выполнить поиск — запросы, которые в традиционной базе данных могут занимать секунды, возвращаются за десятки миллисекунд.

60K

События в секунду

Индексирует кластер из трех узлов

1K+

Поисковых запросов

Обрабатывает одновременно в секунду

Типичный кластер Elasticsearch из трех узлов может индексировать около 60 000 событий в секунду, одновременно обрабатывая более 1000 поисковых запросов в секунду. Компромисс заключается в том, что индексирование происходит медленнее, чем простая вставка в базу данных, и вы получаете конечную согласованность — результаты поиска могут отставать от самых последних данных.

Специализированные решения: оптимизированные для домена модели данных

TimescaleDB: временные ряды в PostgreSQL

TimescaleDB доказывает, что иногда лучшая инновация — это понимание, когда не нужно изобретать велосипед. Вместо того чтобы создавать еще одну базу данных временных рядов с нуля, разработчики расширили PostgreSQL, добавив автоматическое разделение по времени. Вы получаете всю привычную мощь SQL, а также оптимизации, специально разработанные для рабочих нагрузок временных рядов.

Гениальность заключается в «гипертаблицах» — то, что для вашего приложения выглядит как одна таблица, на самом деле автоматически разбивается на фрагменты по времени. Недавние данные хранятся в быстрых, несжатых фрагментах для быстрой записи, а более старые данные сжимаются для эффективного хранения. Вы пишете стандартный SQL, но получаете производительность временных рядов.

TimescaleDB демонстрирует, как специализированные базы данных могут использовать существующие технологии. Созданная как расширение PostgreSQL, она автоматически разбивает данные временных рядов на фрагменты на основе временных интервалов. Недавние данные хранятся в быстрых, несжатых фрагментах, а более старые данные сжимаются для эффективного хранения.

Этот гибридный подход идеально подходит для систем мониторинга. Cloudflare использует TimescaleDB для обработки около 100 000 агрегированных метрик в секунду, одновременно обслуживая сложные запросы с временными интервалами за миллисекунды. Основа PostgreSQL означает, что вы получаете полные возможности SQL с оптимизацией для временных рядов.

100K

Метрик в секунду

Cloudflare обрабатывает в TimescaleDB

Графовые базы данных: отношения на первом месте

Графовые базы данных рассматривают отношения как первостепенный элемент. Вместо моделирования связей с помощью внешних ключей и соединений, они хранят отношения непосредственно в виде ребер между узлами. Когда данные в основном касаются связей — социальные сети, системы рекомендаций, обнаружение мошенничества — эта модель кажется естественной.

Команды часто сталкиваются с трудностями при выражении в SQL запроса «найти друзей друзей, которым нравятся похожие фильмы», для чего требуются сложные рекурсивные CTE, которые работают неэффективно. Тот же запрос на языке Cypher в Neo4j читается почти как английский и выполняется эффективно, поскольку база данных оптимизирована для обхода отношений.

Графовые базы данных, такие как Neo4j, оптимизированы для запросов по отношениям. Традиционные конфигурации реплицируют весь граф в каждый экземпляр, ограничивая масштабируемость записи, но позволяя масштабировать чтение линейно. Новая функция Fabric в Neo4j пытается разбивать графы на кластеры, хотя это остается сложной задачей, поскольку отношения естественным образом выходят за пределы границ.

Результат проявляется в возможностях запросов. Поиск взаимных связей в социальной сети может потребовать сложных соединений в реляционной базе данных, но в графовой базе данных это становится простым переходом, который часто выполняется менее чем за 50 мс.

50

Миллисекунд

Время выполнения поиска взаимных связей

OLAP-решения: аналитика в больших масштабах

OLAP-системы представляют собой фундаментальный сдвиг в нашем представлении о базах данных. Вместо оптимизации для поиска отдельных записей, как в OLTP-системах, они предназначены для эффективного сканирования и агрегирования миллионов строк. Это отражено в модели данных — все структурировано так, чтобы ускорить аналитические запросы, даже если для этого приходится жертвовать транзакционными функциями.

Переход к столбцовому хранению стал прорывом. Вместо хранения строк вместе (где вы читаете много нерелевантных данных), столбцовые системы хранят каждую колонку отдельно. Когда вы хотите суммировать доход по миллионам транзакций, вы читаете только колонку дохода, а не имена клиентов, адреса или любые другие поля. Одно только сокращение объема ввода-вывода может ускорить запросы в 10 раз.

SQL в этих системах выглядит знакомо, но имеет интересные расширения. Большинство систем OLAP поддерживают стандартный SQL с аналитическими расширениями — мощными функциями окон, статистическими операциями и анализом временных рядов, встроенными прямо в язык запросов. Некоторые системы, такие как ClickHouse, имеют собственный диалект SQL с дополнительными функциями, оптимизированными для аналитики, а другие, такие как BigQuery, расширяют стандартный SQL за счет обработки массивов и вложенных данных. Он по-прежнему декларативен, но оптимизирован для сложных аналитических запросов, которые были бы проблематичны в традиционных системах OLTP.

ClickHouse: мощный инструмент для аналитики

ClickHouse представляет собой другой подход к масштабируемости — он оптимизирован специально для аналитических нагрузок, а не для транзакционных. В то время как традиционные базы данных превосходны в обработке отдельных записей, ClickHouse предназначен для сканирования и агрегирования миллионов строк за секунды.

Архитектура построена на основе столбцового хранения и векторного выполнения запросов. Вместо хранения строк вместе, ClickHouse хранит каждую колонку отдельно, что позволяет читать только те колонки, которые необходимы для запроса. Это значительно сокращает ввод-вывод для аналитических запросов. Когда вам нужно суммировать доход по миллионам транзакций, ClickHouse читает только колонку дохода, игнорируя имена клиентов, адреса и другие нерелевантные данные.

2B	200
Строк в секунду	Миллисекунд
Один сервер ClickHouse может обрабатывать для простых агрегаций	Время сканирования сотен миллионов записей журнала

Показатели производительности впечатляют. Один сервер ClickHouse может обрабатывать более 2 миллиардов строк в секунду для простых агрегаций. Сложные аналитические запросы, которые в традиционных базах данных могут занимать несколько минут, часто выполняются за секунды. В реальных условиях развертывания отмечается сканирование сотен миллионов записей журнала и возвращение результатов менее чем за 200 мс.

ClickHouse масштабируется горизонтально с помощью шардинга, но с оптимизацией для аналитики. Каждый шард содержит поднабор данных, а запросы распределяются по шардам для параллельной обработки. Система автоматически обрабатывает сложность распределенных агрегаций, объединяя результаты из нескольких шардов в окончательные ответы.

Компромиссом является специализация. ClickHouse не предназначен для высокочастотных обновлений или сложных транзакций. Он отлично подходит для рабочих нагрузок, где данные постоянно добавляются и выполняются аналитические запросы. Типичным сценарием является потоковая передача событий из Kafka непосредственно в ClickHouse для панелей мониторинга аналитики в реальном времени.

Apache Druid: аналитика в реальном времени

Druid использует другой подход к OLAP, уделяя особое внимание поступлению данных в реальном времени и выполнению запросов за доли секунды. Он предварительно агрегирует данные во время поступления, сохраняя несколько сводных версий ваших данных. Это означает, что некоторые аналитические запросы могут возвращаться мгновенно, поскольку агрегации уже вычислены.

Архитектура разделяет прием данных и запросы. Данные проходят через узлы реального времени, которые обрабатывают входящие потоки, а затем перемещаются в узлы истории, оптимизированные для быстрых запросов. Узлы-брокера координируют запросы по кластеру и кэшируют результаты для еще более быстрых последующих запросов.

Эта конструкция отлично подходит для панелей мониторинга в реальном времени, которые должны отображать метрики, обновляемые в течение нескольких секунд. Рекламные платформы используют Druid для предоставления данных о результативности кампаний в реальном времени, обрабатывая миллионы показов рекламы и обновляя панели мониторинга с задержкой менее 100 мс.

100

Миллисекунд

Задержка обновления панелей мониторинга в реальном времени

Сложность заключается в требованиях к предварительной агрегации. Необходимо определить правила сворачивания во время поступления данных, а это означает, что нужно заранее продумать, какие запросы потребуется запускать. Это хорошо работает для известных аналитических шаблонов, но может ограничивать возможности для спонтанного исследования.

Гибридные стратегии: реальный мир

Большинство успешных систем не полагаются на одну базу данных. Вместо этого они стратегически комбинируют технологии:

Уровни кэширования

Уровни кэширования: Redis или Memcached перед вашей основной базой данных могут поглощать трафик чтения с временем отклика менее миллисекунды. Проблема заключается в инвалидации кэша — поддержании согласованности кэшированных данных с базой данных.

Федерация баз данных

Федерация баз данных: разные типы данных в специализированных базах данных. Профили пользователей в MongoDB для гибкости, финансовые транзакции в PostgreSQL для гарантий ACID и данные поиска в Elasticsearch для полнотекстовых запросов.

Материализованные представления

Материализованные представления: предварительно вычисляйте дорогостоящие запросы и храните результаты в таблицах с быстрым доступом. Это может превратить аналитические запросы, занимающие секунды, в поиски, занимающие 10–100 мс, хотя обновление представлений добавляет сложность.

Крупный сайт электронной коммерции может использовать Redis для кэширования сеансов (миллисекундная скорость), PostgreSQL для обработки заказов (соответствие ACID) и Elasticsearch для поиска продуктов (полнотекстовые возможности). Каждая система выполняет то, что у нее получается лучше всего.

Путь вперед

Масштабируемость баз данных не заключается в поиске «лучшей» базы данных — она заключается в сопоставлении моделей данных и стратегий масштабирования с рабочими нагрузками. Система IoT с интенсивной записью требует другой гибкости модели данных и других возможностей масштабирования, чем аналитическая платформа с интенсивным чтением и сложными требованиями к запросам.

Современные облачные базы данных упрощают эту задачу. Такие сервисы, как AWS Aurora Serverless v2, Google Spanner и Azure Cosmos DB, автоматически масштабируются в фоновом режиме, взимая плату на основе фактического использования, а не выделенной емкости.

Ключ к успеху — понимание ваших конкретных требований: какой объем данных у вас будет? Каково соотношение чтения и записи? Насколько структурированы ваши данные? Насколько должны быть согласованы ваши данные? Какая сложность запросов вам нужна? Какова ваша допустимая задержка?

01

Начните с простого

Начните с простой настройки, измерьте реальную производительность при реалистичных нагрузках

02

Масштабируйте стратегически

Масштабируйте стратегически на основе фактических потребностей

03

Следите за развитием

Среда баз данных продолжает развиваться, но основные принципы остаются неизменными

Среда баз данных продолжает развиваться, но основные принципы — репликация для чтения, фрагментация для записи, кэширование для скорости — остаются неизменными.

> **Помните**: преждевременная оптимизация — корень всех бед, но так же, как и игнорирование масштабируемости, пока не станет слишком поздно. Планируйте рост, но не переусердствуйте с разработкой для проблем, которых еще нет.