УДК 004.42
DOI 10.17513/snt.40173

# ПРОБЛЕМЫ ПРОИЗВОДИТЕЛЬНОСТИ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ В РАСПРЕДЕЛЕННЫХ АРХИТЕКТУРАХ И СТРАТЕГИИ ИХ РЕШЕНИЯ

**Малыгин Д.С.**

*ООО «Т1 Диджитал», Санкт-Петербург, e-mail: malygindms@gmail.com*

Цель исследования состоит в анализе существующих проблем производительности реляционных баз данных, часто встречающихся при создании программных продуктов, использующих микросервисный подход, в рамках распределенной архитектуры. Кроме того, предложены классификация уровней возникновения проблем быстродействия и соответствующие стратегии для их устранения или минимизации негативного эффекта. Для достижения цели был проведен ретроспективный анализ собственного практического опыта создания баз данных для коммерческих программных решений, также изучен опыт зарубежных коллег. Для оценки и демонстрации возможного негативного эффекта от выполнения неоптимального запроса к базе данных был проведен исследовательский эксперимент на реляционной базе с тестовыми данными, продемонстрировавший значительную разницу во времени выполнения между оптимизированным и неоптимизированным запросом, а также временные издержки, которые придется понести в случае операций записи при использовании индексирования таблиц. В качестве результата работы предоставлен ряд рекомендаций, который позволит читателям правильно идентифицировать возникшие проблемы производительности, устранить их последствия, а также избежать фундаментальных ошибок, возникающих на начальных этапах проектирования схемы базы данных, что даст возможность избежать значительных затрат на перепроектирование доменной модели. Работа будет полезна разработчикам программного обеспечения, архитекторам, а также администраторам баз данных.

**Ключевые слова: базы данных, производительность, оптимизация запросов, распределенные архитектуры, микросервисы, индексы, схемы**

# PERFORMANCE ISSUES OF RELATIONAL DATABASES IN DISTRIBUTED ARCHITECTURES AND STRATEGIES FOR RESOLUTION

**Malygin D.S.**

*T1 Digital LLC, Saint-Petersburg, e-mail: malygindms@gmail.com*

The objective of the study is to analyze the existing performance problems of relational databases that are often encountered when creating software products using the microservice approach within a distributed architecture. In addition, a classification of the levels of occurrence of performance problems and appropriate strategies for their elimination or minimization of the negative effect are proposed. To achieve this goal, a retrospective analysis of our own practical experience in creating databases for commercial software solutions was conducted, and the experience of foreign colleagues was also studied. To assess and demonstrate the possible negative effect of executing a non-optimal query to the database, a research experiment was conducted on a relational database with test data, which demonstrated a significant difference in execution time between an optimized and a non-optimized query, as well as the time costs that will have to be sacrificed in the case of write operations when using table indexing. As a result of the work, a number of recommendations are provided that will allow readers to correctly identify the performance problems that have arisen, eliminate their consequences, and avoid fundamental errors that arise at the initial stages of designing a database schema, which will avoid significant costs for redesigning the domain model. The study will be useful for software developers, architects, and database administrators.

**Keywords: databases, performance, query optimization, distributed architectures, microservices, indexes, schemas**

## Введение

Системы хранения информации являются основой современной экономики. По подсчетам, проведенным компанией «Statista», объем глобальных данных к 2025 году превысит 180 ZB (180 триллионов гигабайт) [1]. График на рисунке 1 наглядно показывает, насколько быстро растет объем данных. Несложно предположить путем экстраполяции, что к 2030 году этот объем составит минимум 700–800 ZB. Несмотря на то что лишь небольшая часть из произведенных и потребленных данных сохраняется продолжительное время (примерно 2% [1]), объем сохраняемых данных покажет ежегодный рост в 19,2% [1].

Основная нагрузка, связанная с обработкой настолько большого количества данных, ложится на серверы баз данных. Под нагрузкой понимается набор операций CRUD (Create, Read, Update, Delete) – запись данных, чтение, обновление и удаление. Реляционные базы данных по-прежнему составляют наибольшую часть рынка по сравнению с нереляционными [2].

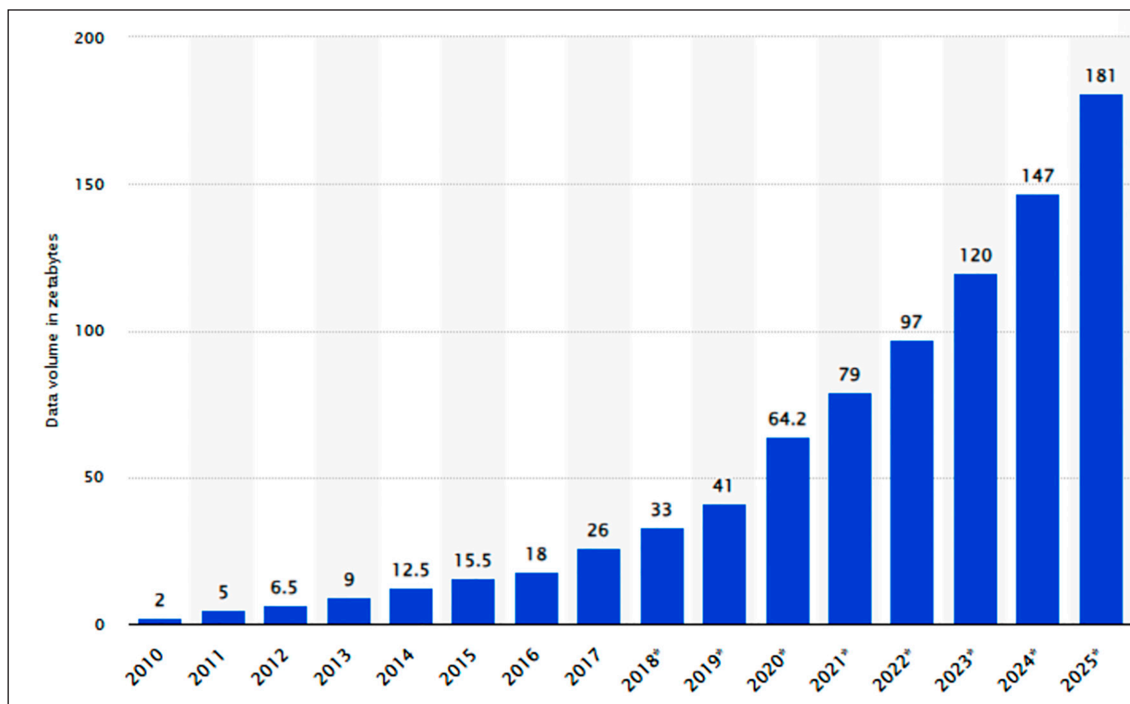


*Рис. 1. Прогнозируемое значение генерируемых данных на 2025 год, зеттабайты (ZB)*

Реляционные базы данных используются повсеместно – от инженерных систем до коммерческих компаний из B2B, B2C секторов. От производительности работы с данными зависит то, насколько эффективно та или иная компания сможет обрабатывать клиентские запросы. Это делает быстродействие систем хранения критически важной характеристикой, которая влияет на бизнес компании. Недостаточная производительность может привести к увеличению времени обработки транзакций, простою системы, что незамедлительно приведет к ухудшению пользовательского опыта. Для коммерческих предприятий это напрямую трансформируется в потерю дохода. Есть несколько довольно красноречивых исследований, когда замедление инфраструктуры, использующей базу данных, может вызвать существенные убытки. Например, результаты A/B тестирования, проведенного компанией Amazon еще в 2006 году, показали, что задержка загрузки страницы лишь на 100 мс может стоить компании 3,8 млрд долларов (по переоценке на 2023 год), или 1% выручки [3]. Другим примером являются расчеты компании Google, показавшие, что замедление поиска всего на 0,4 секунды может снизить количество поисковых запросов примерно на 8 млн в день, что вызовет существенные потери в выручке от показа рекламных интеграций [4].

Теме производительности баз данных посвящен ряд как отечественных, так и зарубежных работ. Например, авторы работы [5] рассмотрели проблематику с трех сторон: предсказание, диагностика и подстройка базы данных к количественным и качественным характеристикам нагрузки обработки данных. Предсказание будущей нагрузки является основой для проектирования системы хранения. Авторы подчеркивают, что, чем качественнее выполнена проработка базы, тем меньше будет стоить устранение проблем в будущем. В работе [6] авторы сконцентрировались на стратегии индексирования, оптимизируя работу базы для функционирования в составе Complex Event Processing (CEP) системы, и предложили новый тип индексации Hierarchical Temporal Indexing (HTI), при этом сравнив его эффективность с традиционными типами индексирования. Авторы в работе [7] провели большой анализ и описали методику повышения скорости выполнения запросов на основе анализа эффективности различных средств, таких как создание индексов, переписывание SQL-запроса, использование материализованного представления, сокращение количества сканирований, и разработали приложение, которое на основе разбора и анализа SQL кода генерирует рекомендации по оптимизации запроса. Сравнение производительности работы сразу нескольких типов баз данных

было проведено в работе [8]. Авторы сравнили быстродействие документно-ориентированной базы MongoDB, графовой – Neo4j, колоночной – Cassandra и объектно-реляционной PostgreSQL баз между собой при обработке таблицы размером 300000 записей. По результатам исследований авторы предложили ряд рекомендаций по выбору базы исходя из поставленных целей.

**Цель данного исследования** – собрать практический опыт решения проблем производительности баз данных на реальных проектах создания сложных программных продуктов, использующих распределенную микросервисную архитектуру, опыт, описанный в релевантных научных статьях, выполнить анализ и предоставить рекомендации, которые будут полезны для инженеров и научных деятелей в области разработки программного обеспечения.

Актуальность данного исследования обоснована активным запросом со стороны как отечественных, так и зарубежных специалистов-практиков из индустрии, решающих задачи построения высоконагруженных систем или их модернизации.

## Материалы и методы исследования

В результате данной работы предложена иерархия уровней возникновения проблем производительности реляционных баз данных, часто встречающихся при построении распределенных архитектур. В работе, преимущественно, рассмотрены архитектуры, использующие микросервисный подход, поскольку он занимает лидирующее положение при разработке современных приложений, применяющихся в коммерческих целях [9]. Кроме того, предложены рекомендации для избежания, устранения либо уменьшения степени деградации быстродействия баз данных. Для выполнения целей исследования были предприняты следующие шаги: *1) выделены критерии для отбора материалов*: тематика – исследования в области производительности баз данных; тип работ – научные статьи, практические задачи (use cases), с которыми пришлось столкнуться автору, а также технические материалы, описывающие опыт мировых ведущих компаний; давность – не более 3–5 лет; *2) проведены анализ и синтез выбранных работ* и выделены наиболее значимые проблемы производительности, оказывающие наихудший эффект на быстродействие; *3) предложена классификация уровней возникновения проблем* производительности на основе детального анализа отобранных случаев; *4) проведен эксперимент на базе данных*, демонстрирующий результаты выполнения SQL-запроса на разных объемах данных – от 100 тыс. записей до 1 млн записей – с учетом узкой выборки и выборки по всем полям таблицы. Также продемонстрированы результаты сравнения времени чтения и записи для индексированного поля. наполненной тестовыми данными для демонстрации размера негативного влияния неоптимизированного SQL-запроса; *5) приведены стратегии и рекомендации для решения проблем с быстродействием базы.* Данное исследование предполагает наличие направлений для более глубокого анализа проблематики в сторону выбора оптимальных технологий хранения данных, программных языков [10] фреймворков, поддерживающих работы с ORM, и архитектурных подходов исходя из количественного анализа влияния на бизнес-показатели компании.

## Результаты исследования и их обсуждение

### *1. Источники возникновения проблем производительности*

Производительность баз данных зависит от множества факторов, начиная от ресурсов CPU, RAM, отведенных на функционирование сервера, заканчивая качеством написанного SQL кода и типом данных. Как правило, проблемы с производительностью возникают на разных уровнях. Для микросервисных архитектур, где каждый модуль использует свою собственную базу данных (в соответствии с лучшими архитектурными практиками [11]), распределенные транзакции представляют собой весьма чувствительную область, которую необходимо мониторить на наличие ошибок и нарушение консистентности данных, вызванные проблемами, появляющимися на уровне SQL-запросов, которые могут выполняться долго, неэффективно, вызывая задержки в выполнении транзакционных изменений. Проблемы SQL-запросов в легких случаях содержатся в самих SQL-запросах, но в тяжелых случаях – это проблемы в самой схеме базы данных. Неправильно спроектированная схема вынуждает писать плохие, неоптимальные SQL-запросы. И если переписать SQL-запрос можно сравнительно просто, то в случае со схемой придется искать обходные, часто лишь временные пути либо перепроектировать состав таблиц.

Конфигурация базы данных тоже может являться порой неочевидным источником проблем. Ресурсы процессора и памяти, к счастью, легко мониторятся специальными инструментами, благодаря чему есть возможность заблаговременно увеличить необходимые ресурсы и хотя бы на вре-

мя отсрочить деградацию производительности. Но поиск причины излишнего потребления ресурсов никуда не исчезает и является задачей для администраторов баз данных. Ниже приведена условная схема, демонстрирующая вышеперечисленные уровни (рис. 2). Описание в статье пойдет от самого верхнего уровня до уровня схемы баз данных, нижние два уровня выходят за рамки данной работы.

## *2. Уровень распределенных транзакций*

Одной из самых непростых задач при построении распределенной архитектуры приложения является обеспечение надежности и безопасности [12] распределенных транзакций – цепочки операций, охватывающих несколько объектов (баз данных). Данный вид транзакций может оказать значительное влияние на производительность, что обусловлено сложностью обеспечения условий атомарности, согласованности, изоляции и надежности (требования ACID) в условиях многокомпонентной архитектуры [13]. На схеме ниже (рис. 3) представлена условная схема приложения, использующего для своей работы несколько баз данных, выполняющих свои отдельные задачи, но которые должны находиться в согласованном состоянии.

Основные причины, из-за которых производительность может снизиться:

• *сетевая задержка* – особенно характерна для георaспределенных кластеров;

• *задержка, связанная с применением двухфазного коммита (Two-Phase Commit (2PC)),* – если одна из баз не посылает сигнал готовности, всю транзакции придется отменять, что вызовет значительные временные расходы;

• *блокировка данных в случае конкурентного доступа* – в том числе и взаимные блокировки (*deadlocks*), касается в первую очередь приложений с высокой частотой транзакций, где параллельные операции часто конкурируют за одни и те же ресурсы;

• *задержки, связанные с репликацией данных.*

*Рис. 2. Уровни возникновения проблем с производительностью баз данных*

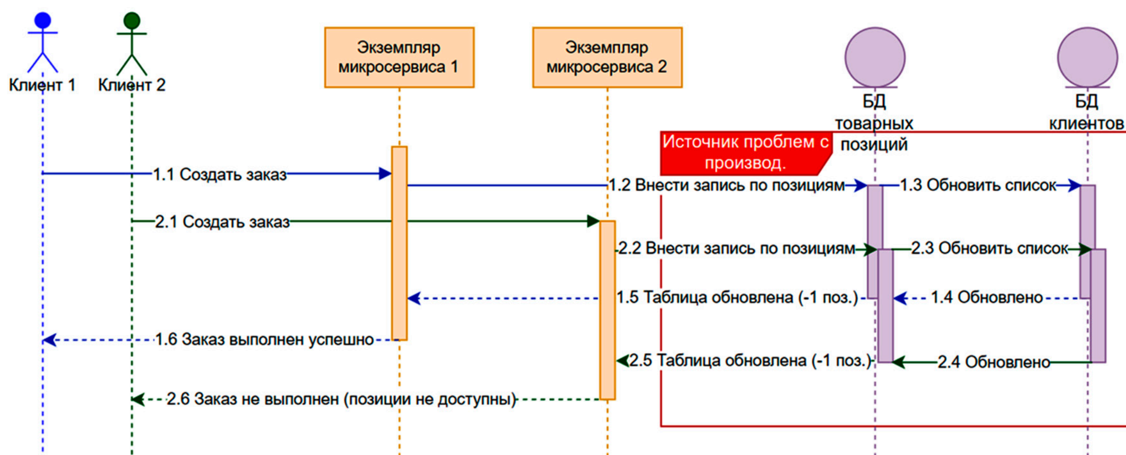


*Рис. 3. Распределенные транзакции – еще один источник проблем с производительностью базы данных*

Одним из очевидных способов решения подобных проблем является сведение потребности в распределенных транзакциях к минимуму путем переработки процессов межмодульного взаимодействия (например, с помощью перераспределения данных). Кроме радикального решения проблем, можно предпринять и ряд других мер, например:

• применить оптимистичную блокировку для ограничения доступа только к узкому набору данных, это может помочь снизить вероятность возникновения взаимных блокировок;

• ввести асинхронную репликацию, если немедленная репликация данных не требуется;

• снизить количество транзакций, объединив их в одну комбинированную (*batching*);

• снизить сетевую задержку за счет оптимизации пути прохождения запроса и устранения элементов, не выполняющих никакой полезной задачи (прокси, шлюзов и т.д.), или за счет более близкого размещения серверов баз данных;

• на регулярной основе проводить мониторинг появления взаимных блокировок и событий в журнале транзакций, а также использовать системы real-time аналитики;

• использовать кэширующий слой для повторяющихся данных.

### *3. Уровень запросов*

#### *3.1. Медленные запросы и их оптимизация*

Другой распространенной причиной проблем производительности в системах управления базами данных является наличие неоптимальных SQL-запросов, которые могут существенно снизить общую эффективность работы системы и привести к значительному расходу ресурсов, таких как процессорное время и оперативная память [14]. SQL-запросы представляют собой набор инструкций, порядок выполнения которых определяется оптимизатором базы данных (рис. 4).

На каждой представленной на схеме ступени возможны подчас довольно очевидные ошибки, исправить которые можно довольно быстро: либо переписав запрос, либо создав индекс. К счастью, такие ошибки не являются дорогими в плане исправления, в отличие от ошибок проектирования схемы базы данных, речь о которых пойдет в следующем пункте.

| Неоптимизированный вариант | Оптимизированный вариант |
|---|---|
| Выполнение поиска по не индексированному полю: *SELECT * FROM users WHERE email_address = 'test@example.com';* – в данном случае, при отсутствии индекса, выполняется полное сканирование таблицы за время O(N) | Создать индекс по искомому полю: CREATE INDEX *idx_email_address* ON users (*email_address*); |
| Выборка всех полей: *SELECT * FROM orders WHERE date = '2020-02-05';* | Уменьшить количество полей в выборке: *SELECT person_name, email_address FROM orders WHERE order_date = '2020-02-05';* |
| Использование подзапроса в операторе IN: *SELECT * FROM customers WHERE customer_id IN (SELECT customer_id FROM orders WHERE order_total > 100);* – подзапрос будет выполняться для каждого условия внешнего запроса | Избегать подзапросов: *SELECT customers.* FROM customers JOIN orders ON customers.customer_id = orders.customer_id WHERE orders.order_total > 100;* |
| Использование дизъюнкции: *SELECT * FROM employees WHERE department = 'logistics' OR department = c OR department = 'legal';* в данном случае обработчику запросов приходится оценивать каждую инструкцию по отдельности, что делает план выполнения неоптимальным | Вместо оператора OR использовать оператор IN: *SELECT * FROM products WHERE department IN ('logistics', 'logistics', 'legal');* |
| Неиспользование операции соединения JOIN: *SELECT customers.name, orders.order_id FROM customers, orders;* что приводит к декартовому произведению, и запрос выполняется за время O(M x N) | Использовать соединение JOIN: *SELECT customers.name, orders.order_id FROM customers JOIN orders ON customers.customer_id = orders.customer_id;* |
| Использование оператора LIKE с подбором по последним знакам шаблона: *SELECT * FROM employees WHERE name LIKE '%Smith';* | Выполнять поиск по начальным символам строки: *SELECT * FROM employees WHERE name LIKE 'Smith%';* |
| Использование функций для индексированных полей: *SELECT * FROM users WHERE UPPER(username) = 'NAME';* – такой запрос лишает преимущества от наличия индекса, заставляя сканировать всю таблицу целиком | Сводить использование вспомогательных функций к минимуму: *SELECT * FROM users WHERE username = 'NAME';* |
| Выполнение действий INSERT / UPDATE по отдельности, а не в составе одной операции | Выполнять операции INSERT / UPDATE в составе одной транзакции |

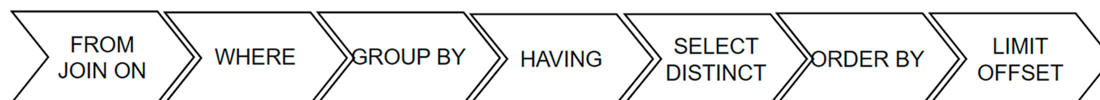


*Рис. 4. Последовательность выполнения инструкций SQL-запроса*

На первый взгляд может показаться, что выборка только по узкому набору полей дает лишь небольшое преимущество в плане быстродействия. Однако результаты эксперимента, описанного ниже, говорят об обратном. Для демонстрации была создана таблица, заполнена разным количеством записей – от 100000 до 1000000, и проведена выборка по всем полям: *SELECT * FROM Performance_test_table. TestItem*: было замерено время выполнения запроса. Затем была произведена выборка по узкому набору полей: *SELECT TestItemOne, TestItemTwo FROMPerformance_test_table.TestItem*. Оба запроса выполнялись на разном количестве записей, с шагом 100000.

По результатам эксперимента (рис. 5) видно, что при небольшом количестве записей (100000) разница не является значительной, но уже при 600 тыс. записей скорость выполнения снижается уже в 3 раза по сравнению с оптимизированной версией запроса. Такое снижение в производительности является неприемлемым, особенно для систем сбора аналитики, систем электронной коммерции, работающих с большим объемом данных.

Основными факторами, которые способствуют ухудшению быстродействия базы данных, являются (особенно в случае баз данных большого объема) неоптимальные SQL-запросы (которые используют полное сканирование таблиц), неэффективные операции объединения, неоптимальное использование индексов и избыточная выборка данных. Введение индексирования определенных полей, оптимизация операций соединения, вставки, обновления позволяют эффективно работать с указанными проблемами и значительно повысить быстродействие.

Создание таблицы:

```
CREATE TABLE [Performance_test_table].[TestItem](
    [TestItemID] [int] IDENTITY(1,1) NOT FOR REPLICATION NOT NULL,
    [TestItemOne] [nvarchar](50) NOT NULL,
    [TestItemTwo] [nvarchar](50) NULL,
    [TestItemThree] [nvarchar](50) NOT NULL,
    [TestItemFour] [nvarchar](50) NOT NULL,
    [TestItemFive] [datetime] NOT NULL
CONSTRAINT [PK_TestItem_TestItemID] PRIMARY KEY CLUSTERED (
    [TestItemID] ASC) ON [PRIMARY])
```

Заполнение таблицы тестовыми данными:

```
DECLARE @i int
SET @i = 0
WHILE @i < 1000000
  BEGIN
    INSERT INTO Performance_test_table.TestItem(
      TestItemOne,
      TestItemTwo,
      TestItemThree,
      TestItemFour,
      TestItemFive
    )
    VALUES(
      CONVERT(nvarchar(50), NEWID()),
      CONVERT(nvarchar(50), NEWID()),
      CONVERT(nvarchar(50), NEWID()),
      CONVERT(nvarchar(50), NEWID()),
      DATEADD (day, (ABS(CHECKSUM(NEWID()))% 1625), GETDATE())
  SET @i = @i + 1
END
```

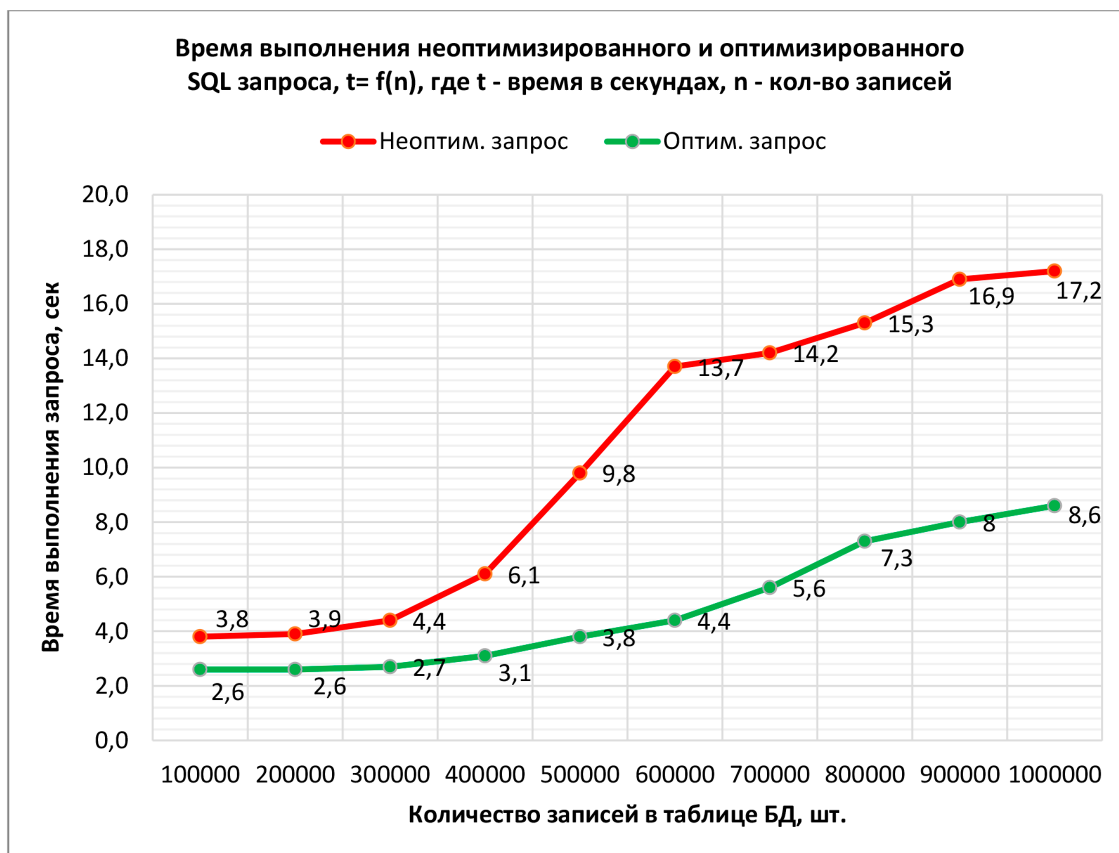


*Рис. 5. Сравнение времени выполнения неоптимизированного и оптимизированного SQL-запроса на чтение*

### *3.2 Проблемы индексации*

Отсутствие индексов или неверно подобранная стратегия индексирования записей в таблице базы данных являются одними из основных факторов, способствующих снижению производительности системы. Разработка эффективной индексации играет ключевую роль в повышении быстродействия, особенно в условиях увеличения объема и сложности данных.

Основными причинами, из-за которых снижается производительность, являются:

• отсутствие индексов для часто запрашиваемых столбцов;

• чрезмерная индексация (слишком много индексов), которая замедляет операции записи;

• использование неселективных индексов, которые не повышают производительность;

• фрагментированные индексы, требующие регулярного обслуживания.

При проектировании той или иной системы необходимо сразу определять спецификацию нагрузки на хранилище данных – OLAP (*Online analytical processing*), OLTP (*Online transaction processing*). Данные спецификации определяют два сценария использования:

• приложения, которые часто считывают большие объемы данных, такие как системы отчетности: BIP (*Business Intelligence Platforms*); ERP (*Enterprise Resource Planning*); *Customer Relationship Management* (CRM) и прочие классы систем для анализа данных;

• системы с высоким уровнем транзакций, которые требуют баланса между чтением и записью: E-commerce платформы; системы резервирования (*Booking, Uber*); системы для трейдинга (E*TRADE) и прочие, классы систем, часто меняющие данные в базах.

Для демонстрации катастрофического эффекта в снижении производительности базы данных было выполнено два запроса – поиск по определенному полю: без использования индекса и с его использованием; другой запрос – вставка в таблицу без индекса и с индексом. Оба запроса были проведены к таблице, определение которой было упомянуто выше. Для сравнения сразу рассматривались два случая – 100 тыс. и 1000 тыс. записей в таблице (рис. 6).

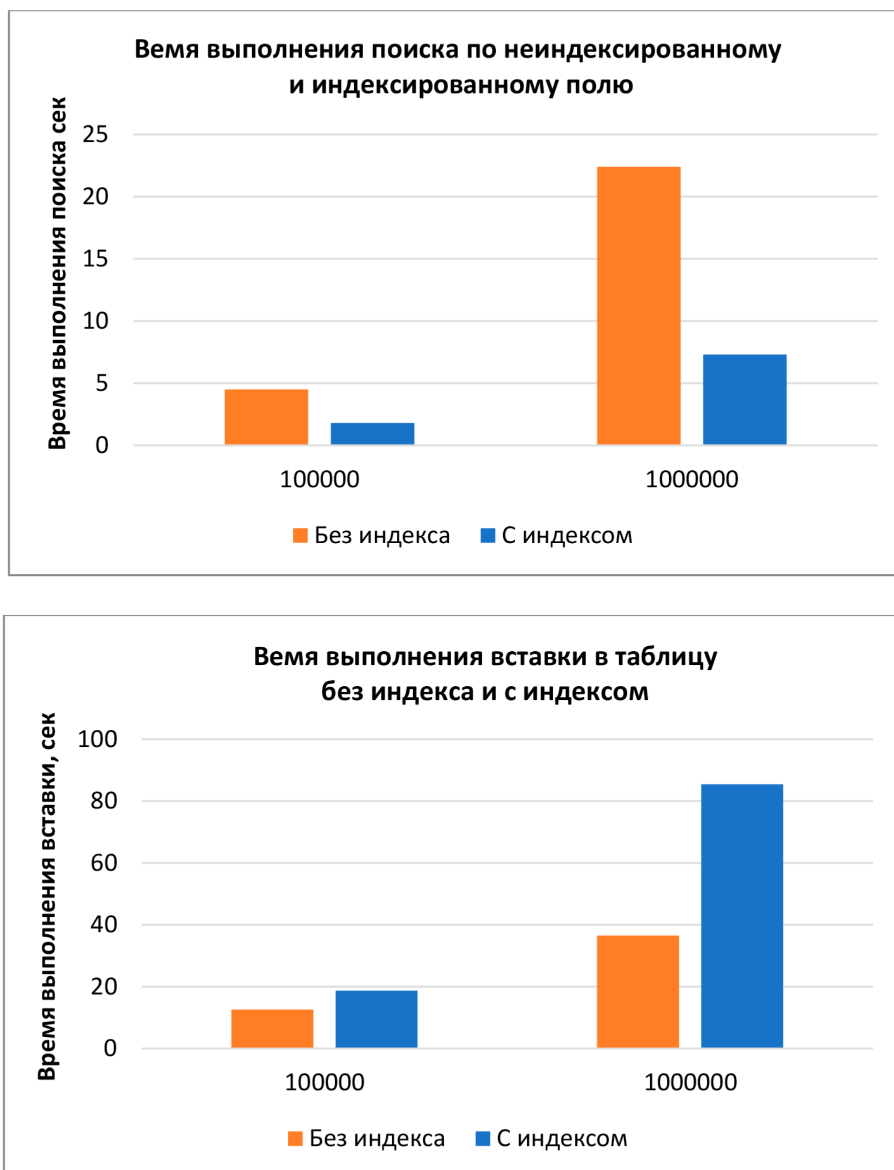


*Рис. 6. Сравнение производительности выполнения операций выборки (SELECT) и записи (INSERT) по индексированному полю и неиндексированному, для 100000 и 1000000 записей*

При планировании использования индексов необходимо опираться на cost&benefit анализ – как видно из второго графика, наличие индекса может стоить довольно дорого, если приложению придется вносить записи в таблицу. В некоторых случаях приходится и вовсе отказываться от использования индексов, жертвуя скоростью поиска в угоду скорости вставки. Индексы требуют тщательного планирования и сбалансированного подхода: отсутствие индекса для внешнего ключа может привести к полному сканированию таблиц во время соединений, что экспоненциально увеличивает время выполнения, однако добавление индексов способно значительно снизить производительность.

Выявление проблем с быстродействием происходит с использованием все тех же инструментов, упомянутых ранее: поставляемых вместе с самими базами данных, такими как pgAdmin, SQL Server Profiler, MySQL Enterprise Monitor, или отдельных продуктов, например как SolarWinds Database Performance Analyzer. Следует придерживаться следующих возможных действий для устранения уже возникших проблем (но все же лучше их избегать на этапе проектирования, поскольку стоимость ошибки растет со временем от старта проекта):

• собрать статистику и определить избыточные или неиспользуемые индексы:

□ прежде всего, стоит проанализировать статистику использования тех или иных индексов с помощью встроенных утилит, предоставляемых базой данных, например: *sys.dm_db_index_usage_stats* в SQL Server, *EXPLAIN* в MySQL или *pg_stat_user_indexes* в PostgreSQL;

□ как правило, введения индексации требуют поля, часто использующиеся в функциях: WHERE, JOIN или ORDER BY; если имеются индексы на полях, не участвующих в данных операциях, то стоит задуматься об удалении/оптимизации таких индексов, если не предусмотрено иное;

□ если имеются индексы на тех же самых полях, то стоит рассмотреть возможность объединения таких индексов в композитный индекс (composite index);

• проанализировать паттерны запросов:

□ необходимо помнить, что архитектура программного продукта должна учитывать почти все возможные изменения, в данном случае: количество данных и характер запросов. Один и тот же индекс может быть уже неэффективным и даже наносить вред при увлечении данных и/или при изменении запросов;

□ необходимым подходом может быть пересмотр настроек батчинга запросов на запись (*batching*) – объединение нескольких операций записи в одну транзакцию существенно снижает нагрузку на сервер БД и повышает быстродействие;

□ если операции записи выполняются по расписанию (*scheduled operations*), то имеет смысл отключить индексацию на время записи, а после – активировать ее вновь и перестроить индексы, такая тактика особенно актуальна в ночное время, когда нагрузка на систему минимальна;

• удалить индексы, введенные (возможно, по ошибке) для полей с низкой кардинальностью множества значений;

• ввести использование индексов, специфичных для той или иной базы данных:

□ например, для SQL server: «Columnstore Index» (особенно полезен для построения OLAP системы хранения, поскольку из-за особенностей работы позволяет эффективно выполнять большие запросы с агрегированием результатов и соединениями);

• перестроить или реорганизовать индексы, подвергшиеся фрагментации в результате большого количества CRUD операций:

□ с помощью функции *sys.dm_db_index_physical_stats()* определить степень фрагментации индекса;

□ заново перестроить индекс имеет смысл при степени фрагментации более 30%, при степени менее данного порога имеет смысл реорганизовать индекс;

• использовать частичные индексы (*Partial index*):

□ если в работе приложения используется лишь часть записей (например, операция чтения выбирает только лиц, проживающих по одному адресу), то имеет смысл заменить индекс, работающий по всей таблице, на индекс, охватывающий только искомое подмножество записей;

• измерить быстродействие операций чтения/записи после внесенных изменений.

Исходя из практики построения сложных распределенных приложений, важными аспектами в обеспечении необходимого уровня быстродействия представляются постоянный мониторинг использования индексов, корректировка стратегий индексирования на основе анализа паттернов запросов с целью предотвращения распространенных ошибок (таких как избыточная индексация или не подходящий набору данных индекс). Эти факторы могут значительно оптимизировать производительность базы данных. Следует помнить, что использовать индексы необходимо с осторожностью, поскольку вместе с оптимизацией они могут произвести значительный негативный эффект на производительность базы данных, особенно в средах с высоким объемом записи.

### *4. Уровень схемы БД*

Схема базы данных является самым критичным уровнем, где могут возникнуть причины низкой производительности, поскольку схема – это результат проектирования, которое реализуется в самом начале разработки приложения. Если причины на уровне транзакций, запросов, конфигурации и ресурсов устранить сравнительно несложно, то проблемы уровня проектирования могут повлечь за собой весьма и весьма значительные траты на переделку (здесь автор позволит себе лишний раз продемонстрировать график зависимости затрат от времени с начала запуска проекта) (рис. 7).

Причинами, связанными со схемой базы данных, вызывающими низкую производительность, могут являться:

• *недостаточная нормализация* – повторяющиеся наборы данных, что вызывает рост таблиц и значительное увеличение времени выполнения операций чтения и записи. Кроме того, увеличивается риск неконсистентности данных при обновлении повторяющихся записей, а также происходит снижение эффективности работы индексов, работа которых сильно зависит от верно подобранной структуры данных;

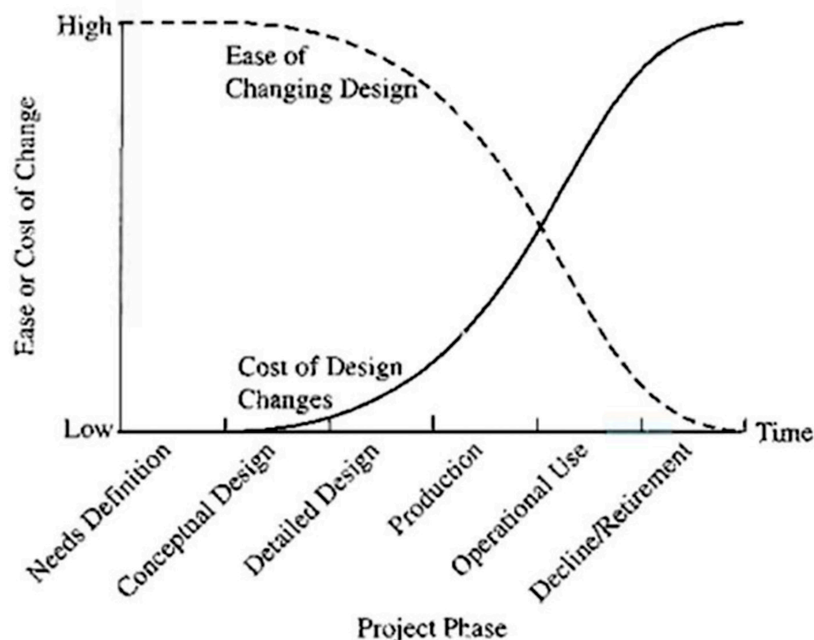


*Рис. 7. Стоимость внесения изменений в систему с течением времени от начала проекта [15]*

• *чрезмерная нормализация* – влечет значительное количество операций соединения (*JOIN*), потребляющих временные ресурсы, а также CPU, RAM. Кроме того, высокая степень нормализации негативно сказывается на времени перестроения индексов;

• *неправильный выбор типов данных* – использование типов «TEXT» или «VARCHAR(255)» для двухбуквенных значений рано или поздно приведет к излишнему потреблению места для хранения данных;

• *отсутствие партиционирования таблиц* – приводит к необходимости сканирования таблиц в полном диапазоне вместо запрашиваемого;

• *наличие цикличных ссылок в реляционном отношении* – приводит к значительному снижению производительности вследствие рекурсивности операций, в частности соединения;

• *отсутствие и/или неправильное использование первичных ключей (primary key);*

• *неправильное использование foreign key;*

• *переизбыток бизнес-логики, реализованной в виде триггеров, хранимых процедур, ограничений,* – схема становится трудномодифицируемой с изменением бизнес-требований.

Лучшим способом для решения вышеупомянутых проблем, конечно, является перепроектирование схемы базы данных с учетом бизнес-требований, их эволюционного изменения, необходимых типов данных и характера запросов. Следует придерживаться следующей тактики для решения таких проблем (возможные действия не расположены в порядке последовательности):

• проанализировать SQL-запросы на уровень сложности – если запрос использует избыточный набор таблиц, возможно, стоит его оптимизировать;

• выполнить валидацию схемы базы данных – с помощью встроенных инструментов SQL Server Data Tools, pgAdmin, MySQL Enterprise Monitor;

• провести аудит использования объектов схемы и выявить неиспользуемые элементы – это можно выполнить с помощью встроенных инструментов, например *pg_stat_user_tables* в PostgreSQL или Dynamic Management Views в SQL Server;

• сбалансировать уровень нормализации базы данных, разнеся данные по логическим группам – отдельным таблицам, но при этом избегая излишнего разбиения;

• проводить регулярный замер элементов базы данных, в частности увеличение размера тех или иных таблиц, которые потенциально могут замедлить выполнение запросов.

Выявление проблем, вызванных схемой базы данных, является особенно актуальным для проектов, находящихся в стадии развития и роста клиентской базы, поскольку с корректировкой бизнес-требований и характера запросов текущая схема может оказаться уже неэффективной. Для устоявшихся проектов в случае возникновения проблем можно прибегнуть к вышеперечисленным действиям, но лучшим спосо-

бом будет перепроектирование схемы базы данных. Проблемы, связанные со схемой, часто проявляются в низкой производительности запросов, блокировках, взаимоблокировках или неэффективности обработки больших наборов данных. Использование инструментов мониторинга, специфичных для базы данных, и анализа запросов может помочь точно определить, где схема вызывает проблемы, что позволяет проводить целевые оптимизации для повышения производительности.

**Заключение**

В настоящей работе были определены уровни возникновения проблем с производительностью базы данных – уровень транзакций, уровень SQL-запросов, индексов, конфигурации и уровень ресурсов, требующихся для функционирования хранилища. Были проанализированы первые три уровня, остальные затронуты не были, поскольку конфигурирование является специфичным для того или иного поставщика базы данных, а ресурсное обеспечение – предмет другого исследования. К тому же и конфигурация, и объем памяти, мощность процессора сравнительно просто можно оптимизировать под конкретные нужды.

Распределенные транзакции могут быть источником проблем производительности, в первую очередь, из-за возможных блокировок, вероятность которых необходимо сводить к минимуму, используя оптимистичные блокировки, а также ускоряя операции записи. SQL-запросы являются главным источником низкого быстродействия из-за высокой сложности их конструкций. Необходимо тщательно анализировать и тестировать SQL код, проводя нагрузочные тесты с базой, наполненной расчетным количеством данных, на некоторую степень, превышающую количество, планируемое на производственном режиме (production environment). Проблемы, возникшие на уровне схемы базы данных, являются самими дорогими в плане устранения, так как для их устранения, в худшем случае, потребуется осуществить перепроектирование базы, что может весьма негативно сказаться на бюджете обслуживания продукта.

Несмотря на обилие разных методов устранения проблем производительности, постоянный мониторинг представляется одним из главных средств профилактики снижения производительности базы данных. Встроенные средства, а также отдельные инструменты диагностики являются полезным подспорьем для поддержания работы базы данных в рабочем состоянии и позволяют устранять возможные проблемы еще до их появления, что может значительно сократить расходы на обслуживание приложения.

**Список литературы**

1. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025. [Электронный ресурс]. URL: https://www.statista.com/statistics/871513/worldwide-data-created (дата обращения: 02.09.2024).

2. Market Share Analysis: Database Management Systems, Worldwide, 2022. [Электронный ресурс]. URL: https://www.gartner.com/en/documents/4432699 (дата обращения: 02.09.2024).

3. Amazon study: Every 100ms in Added Page Load Time Cost 1% in Revenue. [Электронный ресурс]. URL: https://www.conductor.com/academy/page-speed-resources/faq/amazon-page-speed-study (дата обращения: 02.09.2024).

4. How One Second Could Cost Amazon $1.6 Billion In Sales. [Электронный ресурс]. URL: https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales (дата обращения: 09.09.2024).

5. Huang S., Qin Y., Zhang X., Tu Y. Survey on performance optimization for database systems // Sci. China Inf. Sci. 2023. Vol. 66. P. 121102. DOI: 10.1007/s11432-021-3578-6.

6. Abbasi M., Bernardo M.V., Váz P., Silva J., Martins P. Optimizing Database Performance in Complex Event Processing through Indexing Strategies // Data 2024. Vol. 9. P. 93. DOI: 10.3390/data9080093.

7. Бандель Е.С., Нестеров С.А. Методика повышения скорости выполнения запросов в СУБД PostgreSQL // SAEC. 2024. № 2. С. 479-484.

8. Елисеева Е.А., Горячкин Б.С., Виноградова М.В. Исследование производительности СУБД при работе с кластерными базами данных на основе эргономического анализа // StudNet. 2022. № 4. С. 2888-2910.

9. Velepucha V., Flores P. A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges // IEEE Access. 2023. Vol. 11. P. 88339-88358. DOI: 10.1109/ACCESS.2023.3305687.

10. Малыгин Д.С. Тренды и перспективные направления в развитии программирования // Computational Nanotechnology. 2024. Т. 11, № 1. С. 184-192. DOI: 10.33693/2313-223X-2024-11-1-184-192.

11. Abgaz Y., McCarren A., Elger P., Solan D., Lapuz N., Bivol M., Jackson G., Yilmaz M., Buckley J., Clarke P., Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review // IEEE Transactions on Software Engineering. 2023. Vol. 49, No. 8. P. 4213-4242. DOI: 10.1109/TSE.2023.3287297.

12. Maligin D.S. Web Service Availability Monitoring in Distributed Info-communication Systems // Международный научно-исследовательский журнал. 2024. №3 (141). URL: https://research-journal.org/archive/3-141-2024-march/10.23670/IRJ.2024.141.31 (дата обращения: 02.09.2024). DOI: 10.23670/IRJ.2024.141.31.

13. Dragojević A., Narayanan D., Nightingale E.B., Renzelmann M., Shamis A., Badam A., Castro M. No compromises: distributed transactions with consistency, availability, and performance // Proceedings of the 25th Symposium on Operating Systems Principles (SOSP ‘15). Association for Computing Machinery, New York, NY, USA. 2015. P. 54–70. DOI: 10.1145/2815400.2815425.

14. Никишанин Р.О., Ямашкин С.А. Оптимизация SQL-запросов в веб-приложении, написанном на RUBY on RAILS // E-Scio. 2023. №1 (76). URL: https://e-scio.ru/?p=19506 (дата обращения: 02.09.2024).

15. Software Architecture Hidden Costs. [Электронный ресурс]. URL: https://diego-pacheco.blogspot.com/2020/06/software-architecture-hidden-costs.html (дата обращения: 02.09.2024).