10. Nam Mai-Duy, Thanh Tran-Cong. Numerical solution of differential equations using multiquadric radial basis function networks // Neural Networks. 2001. Vol 14. P. 185–199.
11. Li Jianyu, Luo Siwei, Qi Yingjiana, Huang Yapinga. Numerical solution of elliptic partial differential equation using radial basis function neural networks // Neural Networks. 2003. Vol. 16 № 5/6. P. 729–734.
12. Ramuhalli P., Udpa L., Udpa S. Finite Element Neural Networks for Modeling Electromagnetic Forward Problems // AIP Conference Proceedings. 2002. Vol. 615. P. 728–735.
13. Shuixiang Li. Global flexibility simulation and element stiffness simulation in finite element analysis with neural network // Computer methods in applied mechanics engineering. 2000. № 186. P. 101–108.
14. Takeuchi J., Kosugi Y. Neural Network Representation of Finite Element Method // Neural Networks. 1994. Vol. 7. № 2. P. 389–395.

УДК 681.3

МЕТОДЫ ФИЗИЧЕСКОЙ ОРГАНИЗАЦИИ ДАННЫХ, ПОДДЕРЖИВАЕМЫЕ СУЩЕСТВУЮЩИМИ СИСТЕМАМИ УПРАВЛЕНИЯ ДАННЫХ

В. В. ДРОЖДИН, А. М. ВОЛОДИН
Пензенский государственный педагогический университет имени В. Г. Белинского
кафедра прикладной математики и информатики

Рассматривается физическая организация баз данных (БД), создаваемая и поддерживаемая существующими системами управления базами данных (СУБД). Устанавливаются факторы, существенно влияющие на выбор способов организации данных. Определяются методы, позволяющие достичь высокой эффективности обработки данных. Анализируются особенности механизмов хранения данных, поддерживаемых СУБД MySQL.

Для представления и обработки информации об объектах реального мира используются логические и физические структуры данных.

Логические структуры данных (ЛСД) ориентированы на адекватное отражение реального мира и удобное представление информации пользователям. ЛСД носят абстрактный характер (не учитывается их реализация на компьютере), хорошо формализованы и, как правило, имеют точное математическое описание.

Физические структуры данных (ФСД) описывают то, каким образом данные хранятся, как к ним осуществляется доступ и как реализуются операции обработки данных. Поэтому ФСД ориентированы, прежде всего, на эффективную обработку данных.

ЛСД и ФСД, содержащие одну и ту же информацию, часто не совпадают. Поэтому их соответствие осуществляется с помощью отображений "логический – физический" и " физический – логический". Наличие отображений делает ЛСД и ФСД достаточно независимыми и позволяют одну ЛСД реализовать различными ФСД и наоборот. Разница реализаций одной ЛСД различными ФСД заключается в различной эффективности обработки данных.

В данной работе рассматриваются ФСД, поддерживаемые существующими СУБД, и определяются факторы, существенно влияющие на выбор ФСД и эффективность обработки данных.

Для ФСД целесообразно различать логическую и физическую организацию данных. Логическая организация данных (ЛОД) ФСД отражает множество элементов данных и отношения между ними, а также способ реализации операций обработки данных. Физическая организация данных (ФОД) ФСД определяет представление и размещение данных в памяти, метод доступа к данным и механизм реализации операций обработки данных.

На эффективность обработки ФСД существенное влияние оказывают как ЛОД, так и ФОД. Например, для ФСД, являющейся упорядоченным списком, ЛОД может быть задана в виде упорядоченной последовательности блоков, содержащей элементы данных в возрастающем или убывающем порядке, а ФОД – в виде цепочки блоков, связанной указателями, со значениями данных, сжатыми на основе некоторого метода сжатия. Для ФСД, являющейся B^+–деревом, ЛОД определяет многоуровневый индекс и наличие реальных данных на нижнем уровне дерева. При этом в ФОД многоуровневый индекс может быть представлен сжатыми значениями данных, дескрипторами и даже хешированными структурами. Кроме этого ФОД может учитывать организацию и различные характеристики памяти, а также включать дополнительные (управляющие) подструктуры, способствующие повышению эффективности обработки данных. Разнообразие ФОД для оной ЛОД не влияет на логику обработки ФСД, а изменяет только эффективность обработки.

В настоящее время широко известны ФСД, рассмотренные в работах [1-3]. При выборе ФСД решающим фактором является эффективность обработки данных, причем согласно Дж. Мартину [3] в первую очередь обеспечение эффективности поиска, далее идут эффективность операций включения и удаления, а также обеспечение компактности данных. Кроме этого, необходимо учитывать такие факторы, как надежность, изменчивость, взаимосвязь данных, минимизацию избыточности, способность к восстановлению после сбоев и др.

При создании существующих систем разработчики вынуждены принимать компромиссные решения

относительно выбора ФСД. Как показывает практика, некоторые ФСД, хорошо зарекомендовавшие себя при реализации одних ЛСД, оказываются непригодными для реализации других. Методы, обеспечивающие высокую плотность данных, например сжатие данных, приводят к дополнительному усложнению алгоритмов обработки и увеличению времени, затрачиваемого на адресацию и поиск данных в файле.

Средства, обеспечивающие высокую гибкость поиска, как правило, требуют большего объема памяти и большего времени доступа к данным. Увеличение скорости выборки данных часто осуществляется за счет использования более быстродействующих запоминающих устройств и большего объема памяти. Принципиально существует способ организации данных, обеспечивающий минимальное время реализации запроса. Если к некоторым элементам данных происходит частое обращение, то скорость их получения должна быть намного выше по сравнению с данными, которые используются редко. Для этих целей может применяться кэширование и контролируемая избыточность. Для быстрого получения ответа многие системы дублируют наиболее важные элементы данных и помещают их (иногда в очень сжатой форме) в наиболее быстродействующую область памяти. Все эти способы обеспечения быстродействия системы реализуются человеком при проектировании и создании системы.

В системах управления данными часто применяются такие избыточные структуры как индексы, организованные в виде **B- или B$^+$**-деревьев. Индекс представляет собой управляющую структуру данных, основным назначением которой является обеспечение быстрого доступа к данным [3,4]. Они ускоряют процесс сравнения данных, помогают быстро находить минимальное и максимальное значения. Однако деревья перестраиваются по мере вставки и удаления данных. Поэтому каждый индекс снижает скорость добавления, удаления и модификации данных, что приводит к уменьшению производительности.

Принципиально другим способом сокращения времени выдачи ответа является использование адаптивной организации данных, в которой применяется автоматический метод перемещения наиболее часто используемых элементов в такие области памяти, которые обеспечивают возможность более быстрого обращения к этим элементам. Для учета распределения вероятностей использования данных Кнут в [2] рассматривал самоорганизующиеся файлы, а Мартин в [3] приводит адаптивные списковые структуры и метод размещения блоков данных в памяти с различным быстродействием. Однако эти структуры носят преимущественно линейный характер и даже в оптимальном случае не могут конкурировать с иерархическими структурами (например, B$^+$-деревьями). Это объясняется тем, что время доступа к данным и сложность обработки индекса пропорциональны $O(\log_m N)$, где **N – общее число элементов в индексе, а m – число** элементов в блоке. В [4] рассматривается организация многоуровневого иерархического индекса, приспосабливающегося к изменениям интенсивности использования элементов индекса, объема индекса и наличия свободных ресурсов в системе.

ФСД, обеспечивающие высокую эффективность добавления данных в реальном времени, предполагают обработку неупорядоченных данных или требуют использования большего объема памяти для хранения данных. Методы эффективного использования памяти часто оказываются неудовлетворительными в процессе эксплуатации информационной системы, так как приводят к необходимости реорганизации области хранения при добавлении новых данных.

Высокая сложность ФСД иногда может приводить к частичной утрате способности системы к восстановлению после сбоев и потере данных. Поэтому крайне важно, чтобы существовала процедура восстановления тех данных, без которых невозможно дальнейшее функционирование системы. Для обеспечения такой возможности необходимо периодически копировать эти данные в специальную область памяти, а также регистрировать все операции модификации данных. Процедуры рестарта и восстановления после сбоев могут предъявлять специальные требования к физическому размещению данных, вызывая, например, необходимость дублирования части данных.

Размещение данных в памяти компьютера может и должно подвергаться изменениям с целью увеличения оптимальности и эффективности использования данных. Возможны два способа достижения этой цели: параметрическая настройка и периодическая реорганизация физической структуры данных. Системы, способные приспосабливаться к изменениям внешней среды и внутренней организации путем настройки своих параметров или изменения структуры называются адаптивными [5]. Параметрическая адаптация предполагает изменение и фиксацию параметров, определяющих эффективное поведение системы в определенных условиях. Она достаточно проста в реализации, т. к. не требует изменения структуры и законов функционирования системы. Однако она действует в ограниченном диапазоне и не позволяет добавлять новые алгоритмы и структуры данных после создание системы.

Существенно более эффективной является структурная адаптация системы. При этом ЛСД сохраняются, а ФСД заменяются на новые структуры. Механизмы хранения со структурной адаптацией позволяют включать или заменять какие-либо алгоритмические или структурные компоненты другими, позволяющими системе длительное время оставаться адекватной изменяющимся условиям внешней среды. Для достижения максимальной эффективности ФСД следует периодически реорганизовывать, когда система не загружена (ожидает запросов) или имеет незначительную загрузку и в системе остается большой объем свободных ресурсов.

В современных СУБД структуры данных и методы доступа к данным различны и скрыты от пользователя. Способы хранения таблиц базы данных (БД) можно указать только при их создании. В одних СУБД, например MySql (в механизме хранения MyISAM),

данные и индексы физически хранятся в отдельных файлах. В других СУБД, например Oracle, Sybase, InterBase, Firebird, данные и индексы хранятся вместе. В этом случае информация строк таблицы находится на конечных страницах (листьях) индексов. Преимущество такого варианта заключается в уменьшении числа обращений к диску, если индекс хорошо кэширован.

Существуют также СУБД (SDBM, Focus и др.), которые хранят информацию каждого столбца в отдельной области. Такой способ позволяет эффективно использовать методы сжатия данных, но снижается производительность запросов, которые обращаются более чем к одному столбцу.

Широко распространенная СУБД MySql поддерживает несколько механизмов хранения и управления таблицами разных типов [6]. Каждый способ хранения имеет свои особенности и набор параметров конфигурации, с помощью которых можно влиять на производительность. Механизмы хранения данных Berkeley DB, Gemmi и Inno DB ориентированы на обработку данных с помощью транзакций, а механизмы Heap, ISAM, Merge, MyISAM предназначены для обработки таблиц без транзакций, т. е. позволяют использовать произвольные последовательности операций обработки данных.

Транзакционные таблицы более безопасны. Транзакции в большинстве случав реализуются посредством журнальных файлов, куда записываются сведения об изменении табличных данных. В случае возникновения сбоев всегда можно восстановить данные либо из резервной копии и журнала транзакций, либо с помощью автоматического восстановления. Кроме того, механизмы хранения с транзакциями лучше справляются с параллельной обработкой таблиц, в которых одновременно с чтением выполняется обновление данных.

Таблицы без транзакций обладают более высокой скоростью работы и менее требовательны к дисковому пространству, что объясняется отсутствием транзакций.

Тип таблицы указывается при ее создании. Для этого в операторе CREATE TABLE используется опция ENGINE или TYPE:

```
CREATE TABLE ... ENGINE = MyISAM;
CREATE TABLE ... TYPE = MyISAM;
```

Для того, чтобы в дальнейшем преобразовать таблицу из одного формата в другой можно воспользоваться оператором:

```
ALTER TABLE ... TYPE = InnoDb;
```

В этом случае будет создана новая таблица с другим механизмом хранения данных, в которую будут импортированы данные из исходной таблицы. После завершения операции исходная таблица удаляется. Администратор базы данных (АБД) может также создать пустую таблицу со схемой данных как в исходной, но с другим форматом хранения, и вставить в нее строки с помощью команды

```
INSERT INTO ... SELECT * FROM ...
```

Подробное описание механизмов хранения MySql можно найти на официальном сайте www.mysql.com, а также в технической документации, справочниках, руководствах и статьях. В большинстве этих механизмов индексы представлены в виде B- или B^+-деревьев. Для ускорения поиска в индексе в MySQL могут использоваться хешированные индексы. Хешированные индексы создаются либо для всего индекса B-дерева, либо для его части. Так, например, в механизме InnoDB используются адаптивные хешированные индексы, которые создаются по запросу для тех страниц индекса, к которым часто производится доступ. Хотя механизм адаптивного хешированного индекса InnoDB приспосабливается к большому объему данных, он больше подходит для БД, размещаемых в основной памяти [6].

Данные в MySQL хранятся либо в отдельных файлах данных (MyISAM), либо в листовых страницах индексов (BerkeleyDB, InnoDb). Механизм MyISAM позволяет разделить данные и индексы, однако при большом количество таблиц необходимы некоторые накладные расходы. Операции создания, открытия и закрытия таблиц будут более медленными, т. к. для каждой таблицы, которая должна быть открыта, другая должна быть закрыта. Эти расходы можно снизить, увеличив размер табличного кэша. В MyISAM данные могут храниться в сжатом формате и занимать небольшое дисковое пространство, что позволяет сократить число обращений к диску, но они предназначены только для чтения.

В механизме BerkeleyDB для хранения данных применяются четыре метода доступа: Hash, BTree, Queue и Recno. Чаще всех используется метод BTree, основанный на упорядоченном по ключу, сбалансированном B^+-дереве. Данный метод обладает всеми преимуществами B^+-деревьев, но его повсеместное использование не всегда оправдано. На небольшом наборе данных он имеет примерно такую же скорость доступа, что и метод Hash, реализующий расширяемую линейную хеш-таблицу. С ростом объема данных эффективность метода BTree снижается, особенно при большом числе произвольных запросов. Это связано с тем, что в методе BTree требуется больше служебной информации, поэтому кэш быстрее заполняется и требуется большее число операций ввода-вывода. Особенность метода Hash заключается в том, что хеш-функция адаптируется по мере роста объема данных, и все хеш-корзины в устойчивом состоянии по мере возможности остаются заполненными не до конца. К недостаткам метода Hash можно отнести то, что он поддерживает возможность поиска лишь по точному совпадению данных. Хеш-индексы работают очень быстро, но используются только для сравнений на равенство и неравенство (= и < >).

Методы доступа Queue и Recno являются эффективными в случае использования логических номеров записей в качестве первичного ключа таблицы. Они построены на базе метода доступа B^+-дерева и предоставляют простой интерфейс для хранения значений, упорядоченных по номеру. Метод досту-

па Queue позволяет эффективно управлять последовательностью записей фиксированной длины. Он использует механизмы блокировки и протоколирования, оптимизированные для работы в условиях интенсивного многопользовательского доступа к голове и хвосту очереди.

Метод Recno позволяет обрабатывать данные, хранимые в виде записей фиксированной и переменной длины. Он автоматически генерирует номера записей, начиная с единицы. Имеется возможность задать системе автоматическую перенумерацию записей после добавления или удаления записей с меньшими номерами, что позволяет поддерживать плотный список номеров и вставлять данные между существующими записями.

Таким образом, в современных СУБД используются методы доступа, организованные преимущественно на основе B- или B^+-деревьев. Использование деревьев обусловлено предположением равновероятного доступа к любым данным и незначительным ростом высоты дерева при увеличении количества данных. Как известно, в B-деревьях время доступа к данным и сложность их обработки имеют логарифмическую зависимость, что дает им преимущество по сравнению с линейными структурами данных. Однако часто такое предположение неверно. Поэтому в механизмах хранения применяются комбинированные методы доступа, использующие преимущества других структур данных: множеств, последовательностей, хеш-таблиц и т. д. Для достижения высокой производительности необходимо явным образом задавать (настраивать) параметры конфигурации. АБД может изменять размеры страниц, файлов данных, буфера (кэша) и др. параметры, специфичные для каждой ФСД. Замена одних структур другими возможна, но она также выполняется человеком. Для этого создается новая ФСД, в которую копируются данные из исходной структуры. После чего исходная структура удаляется.

СПИСОК ЛИТЕРАТУРЫ

1. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. 536 с.
2. Кнут Д. Искусство программирования на ЭВМ: сортировка и поиск. Т. 3. М.: Мир, 1978. 844 с.
3. Мартин Дж. Организация баз данных в вычислительных системах. М: Мир, 1980. 662 с.
4. Дрождин В. В. Организация адаптивного индекса // Математика и информатика: Межвузовский сборник. Пенза: ПГПУ, 1996. С. 80–85.
5. Ордынцев В. М. Системы автоматизации экспериментальных научных исследований. М.: Машиностроение, 1984. 328 с.
6. MySQL. Руководство администратора. М.: Издательский дом «Вильямс», 2005. 624 с.

УДК 681.3

МОДИФИКАЦИЯ СИСТЕМЫ SQL-ЗАПРОСОВ ПРИ ИЗМЕНЕНИИ ПОЛЬЗОВАТЕЛЬСКОГО ОБЪЕКТНОГО ПРЕДСТАВЛЕНИЯ ПРЕДМЕТНОЙ ОБЛАСТИ

В. В. ДРОЖДИН, Р. Е. ЗИНЧЕНКО
Пензенский государственный педагогический университет имени В. Г. Белинского
кафедра прикладной математики и информатики

Рассмотрена проблема изменения системы базовых SQL-запросов, поддерживающих объектное представление предметной области, при изменении пользовательских представлений и предложен алгоритм автоматической генерации базовых запросов.

При создании автоматизированных информационных систем (АИС), способных самостоятельно поддерживать внешние (пользовательские) представления предметной области (ПО) необходимо дать системе метод (алгоритм), с помощью которого она сможет автоматически приводить в корректное состояние структуру базы данных (БД) и систему базовых SQL-запросов, отображающих пользовательские представления объектов ПО в БД.

В качестве примера рассмотрим предметную область “Университет”, включающую специальности, которым обучаются в университетах. Предположим, что абитуриент решил поступить на математическую специальность некоторого вуза. Поэтому для данного абитуриента имеет ценность информация о вузах и предлагаемых ими математических специальностях. Внешнее представление пользователя состоит из двух сложных понятий “вуз” и “специальность”, содержащих элементарные (простые) понятия: название, адрес, ректор, код, квалификация, и выглядит следующим образом:

вуз (название, адрес, ректор)

специальность (вуз, название, код, квалификация)

При этом “название” в объекте “вуз” является название вуза, а “название” в объекте “специальность” – название специальности. Свойство (атрибут) “вуз” в объекте “специальность” является ссылкой на объект “вуз” и указывает, что все свойства объекта “вуз” включаются в объект “специальность”.

Построим информационную модель ПО в форме ER-модели.

Примем соглашения: 1) название первичного ключа отношения должно совпадать с названием