РОССИЙСКО-АРМЯНСКИЙ (СЛАВЯНСКИЙ) УНИВЕРСИТЕТ

Геворгян Григор Рубенович

ОБ ОДНОМ ПОДХОДЕ К ИНТЕГРАЦИИ ДАННЫХ: МОДЕЛЬ, АЛГОРИТМЫ И ВЕРИФИКАЦИЯ

Диссертация

на соискание ученой степени

кандидата технических наук

05.13.04 – «Математическое и программное обеспечение вычислительных машин, комплексов, систем и сетей»

Научный руководитель

д. ф.-м. н., профессор Шукурян С. К.

Ереван-2016

Содержание

ВВЕДЕНИЕ

Актуальность проблемы

Целью интеграции данных является использование содержимого двух или более источников данных (баз данных) в рамках более крупной, возможно виртуальной, базы данных с целью обращения к ней с запросами как к унифицированному информационному пространству. Интеграционная система должна предоставлять пользователю унифицированный взгляд на множество неоднородных источников информации, называемый *глобальной схемой*.

Основными технологиями, используемыми при решении задачи интеграции данных, являются *федеративные базы данных*, *медиаторы* и *хранилища данных*. При использовании федеративных баз данных источники информации независимы, но каждый из них способен получать требуемую информацию из других. Примерами современных приложений, использующих технологии федеративных баз данных, являются IBM DB2, Oracle Data Integrator и GoldenGate.

Медиаторы представляют из себя программные компоненты, обеспечивающие поддержку виртуальных баз данных (virtual databases), которые “внешне” выглядят так, словно они материализованы (т.е. сконструированы физически, подобно хранилищам данных). Медиатор сам по себе не сохраняет информацию – вместо этого он транслирует запрос пользователя в один или несколько запросов, адресованных первичным источникам. Получая результаты обработки частных запросов, медиатор синтезирует на их основе ответ на исходный запрос. Системы интеграции данных формально определяются в виде тройки $\langle G, S, M \rangle$, где G – глобальная схема (схема медиатора), S – множество схем источников, а M – отображение между запросами над глобальной схемой и схемами источников. В архитектурах, основанных на идее медиатора, именуемых также архитектурами с посредником, используются три разновидности представления интегрированных данных – Global as View (GAV), Local as View (LAV) а также их совмещение, именуемое GLAV. Согласно GAV [1], [2], глобальная схема определяется как взгляд над источниками данных, в то время как LAV [1], [2]

предполагает, что источники данных определены как взгляд над глобальной схемой. Подход GAV имеет лучшую производительность обработки запросов чем LAV, но LAV имеет лучшую расширяемость чем GAV. Для совмещения преимуществ упомянутых технологий был предложен смешанный подход [3], называемый GLAV. Существует метод компиляции системы GLAV в эквивалентную ей систему GAV [4]. Подход к интеграции данных, предлагаемый в данной работе, относится к разновидности GLAV. Среди современных исследований в области виртуальной интеграции данных следует выделить работы группы SYNTHESIS [5], [6], которые являются пионерами в области исследования методов интеграции неоднородных источников информации. Группой SYNTHESIS были предложены концепция канонической модели данных и принцип коммутативных отображений моделей данных [7], который является основой предлагаемого в данной работе подхода к интеграции данных. Также следует выделить исследования Паоло Атцени [8], [9], которые ориентированы на задачи интеграции как структурированных, так и NoSQL баз данных.

При использовании хранилищ данных копии фрагментов информации из нескольких источников сохраняются в единой базе данных, возможно, с предварительной обработкой – *фильтрацией* (filtering), *соединением* (joining) или *агрегированием* (aggregating). Содержимое хранилища, как правило, обновляется на периодической основе. В процессе копирования данные подвергаются определенным преобразованиям с целью согласования их структур с общей схемой хранилища. Основным недостатком хранилищ данных является их потенциальное несоответствие реальному времени, так как изменения в источниках данных могут не быть своевременно в них отражены [10]. Важными классами приложений систем интеграции информации, основанных на технологии хранилищ данных, на сегодняшний день являются приложения *OLAP* (от on-line analytical processing – оперативная аналитическая обработка данных) и приложения *интеллектуального анализа данных* (data mining).

Возникновение новой парадигмы в науке и различных приложениях информационных технологий связано с проблемами обработки *больших данных* (big

data). Концепция больших данных относительно нова, и следующие пять характеристик обычно используются в качестве определяющих: объем (volume), скорость (velocity), многообразие (variety), достоверность (veracity) и значимость (value) [11]. Одной из ключевых задач в данной области является интеграция неоднородных источников информации.

Цель и задачи работы

Целью диссертационной работы является исследование задач интеграции данных, разработка подхода к интеграции данных, а также алгоритмов поддержки виртуальной и материализованной интеграции.

Достижение цели предполагает решение следующих задач:

1. Разработка формальных основ подхода к интеграции данных;
2. Разработка медиатора для поддержки виртуальной интеграции данных;
3. Разработка хранилища для поддержки материализованной интеграции данных.

Методы исследования

При решении поставленных в работе задач использовались методы Нотации Абстрактных Машин (Abstract Machine Notation – AMN) [12], теории множеств и логики предикатов.

Научная новизна

В диссертационной работе получены следующие результаты:

1. Разработан подход к интеграции данных на основе принципа коммутативных отображений моделей данных;
2. Предложена расширяемая XML-каноническая модель данных;
3. Построены AMN-машины для канонической и реляционной моделей данных и их обратимого отображения и доказана его корректность;

4. Разработан подход к построению хранилища данных, основанный на динамической структуре индекса для многомерных данных, предложены эффективные алгоритмы для ее поддержки и приведены оценки сложности предложенных алгоритмов;
5. Разработан и реализован прототип хранилища данных на основе предложенной динамической структуры индекса.

Практическое значение

Предлагаемый в работе подход к интеграции данных может быть использован для интеграции неоднородных источников информации в приложениях аналитической обработки в режиме реального времени, приложениях интеллектуального анализа данных, а также в области больших данных. Эксперименты, проведенные с использованием разработанного прототипа хранилища, показывают, что использование предложенного подхода может привести к значительному уменьшению размера директории индекса и времени поиска для многомерных данных.

Положения, выносимые на защиту

1. Расширяемая XML-каноническая модель данных;
2. Подход к виртуальной интеграции данных;
3. Подход к материализованной интеграции данных;
4. Прототип хранилища данных на основе предложенного подхода к материализованной интеграции данных.

Апробация работы

Основные результаты диссертационной работы обсуждались на семинарах кафедры системного программирования Российско-Армянского (Славянского) Университета (РАУ) и кафедры информационных систем Образовательного и исследовательского центра информационных технологий Ереванского Государственного Университета (ЕГУ). Основные результаты диссертационной работы

докладывались на симпозиуме First Workshop on Programming the Semantic Web в рамках конференции International Semantic Web Conference 2012г, а также на годичных научных конференциях РАУ 2011, 2012, 2015гг.

Публикации

Основные результаты исследований отражены в 6 научных публикациях [13], [14], [15], [16], [17], [18].

Структура и объем работы

Диссертация состоит из введения, четырех глав, заключения и списка использованной литературы (92 наименования). Общий объем диссертации – 102 страницы.

Благодарности

Автор выражает глубокую благодарность к.ф.-м.н., доценту М. Г. Манукяну за неоценимые советы, поддержку и помощь в работе на протяжении всех этапов исследования.

1 ОСНОВНЫЕ ПОДХОДЫ К ИНТЕГРАЦИИ ДАННЫХ

Задачи интеграции данных детально обсуждены в литературе, например в [19], [20]. Интеграционные системы могут обеспечивать интеграцию данных на физическом, логическом и семантическом уровне. Интеграция на физическом уровне подразумевает преобразование данных из источников в требуемый формат их представления. Интеграция на логическом уровне предусматривает возможность доступа к данным разных источников в терминах единой глобальной схемы, описывающей их совместное представление с учетом структурных и поведенческих свойств данных. При этом не учитываются семантические свойства данных. Интеграцию данных с учетом их семантических свойств в контексте единого информационного поля обеспечивает интеграция на семантическом уровне.

Основными подходами к решению задачи интеграции данных являются [10]:

1. *Федеративные базы данных* (federated databases);
2. *Хранилища данных* (data warehouses);
3. *Медиаторы* (mediators);

При выборе любой из этих технологий возникают проблемы, связанные со смысловой интерпретацией данных, поскольку источники информации в общем случае являются неоднородными (heterogeneous). В последующих параграфах данной главы представлены обзоры каждого из этих подходов и связанные с ними технологии.

При интеграции неоднородных источников данных различают следующие основные проблемы:

1. *Различия в типах данных.* Данные имеющие схожую семантику могут храниться в источниках информации в виде объектов различных типов. Например, серийные номера автомобилей могут храниться в одном источнике виде строк переменной длины, а в другом в виде целочисленных значений

2. *Различия в множествах допустимых значений.* Один и тот же атрибут в различных источниках может быть представлен различными наборами констант.

Например, значению черного цвета в одном случае может быть поставлено некоторое целое число, а в другом строка *“black”*.

3. *Семантические различия*. Одно и то же понятие может по разному трактоваться в рамках каждого из источников. Например, отношение *Cars* может представлять из себя как информацию обо всех транспортных средствах, так и только о легковых автомобилях.

4. *Отсутствующие значения*. В некоторых источниках данных может отсутствовать какая-либо информация, присутствующая в других.

При наличии любой из этих ситуаций на начальной стадии процесса интеграции следует предусмотреть и реализовать те или иные механизмы трансляции данных.

1.1 Федеративные базы данных

Федеративные базы данных (federated databases) – источники независимы, но каждый из них способен получать требуемую информацию из других. При этом для каждой пары источников требуется построение транслятора запросов, итоговое количество которых будет порядка $O(n^2)$ при n источниках.

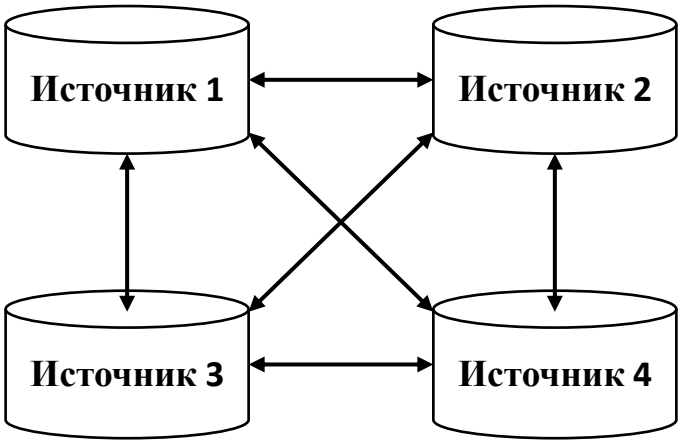


Рис. 1.1. Федеративные базы данных

Примерами современных приложений, использующих, в том числе, технологию федеративных баз данных, являются продукты Data Integrator [21] и GoldenGate [22] компании Oracle. Архитектура Oracle Data Integrator (ODI) организована вокруг модулярного репозитория метаданных, доступ к которому осуществляется в режиме клиент-сервер при помощи специальных компонент. Объекты, разработанные либо сконфигурированные с помощью пользовательских интерфейсов, хранятся в репозиториях. В ODI существует два вида репозиториев: главные репозитории и рабочие репозитории. В платформе ODI может быть сконфигурирован только один главный репозиторий и множество рабочих.

В главном репозитории хранятся следующие метаданные:

- Информация, касающаяся обеспечения безопасности платформы ODI – пользователи, профили и права;
- Информация о топологии – используемые технологии (платформы баз данных), языки, настройки серверов;
- Информация о контроле версий.

Рабочие репозитории хранят следующие метаданные:

- Модели данных – схемы, ограничения и т.д.
- Проекты – интерфейсы, пакеты, процедуры, модули знаний;
- Сценарии выполнения – планы загрузки, расписания, логи и т.д.;

Графический пользовательский интерфейс имеет название ODI Studio, и используется для доступа к главному и рабочим репозиториям. Следующие компоненты, предоставляемые ODI Studio, используются для администрирования инфраструктуры, разработки проектов, планирования и управления системой:

- Designer – компонент ODI в котором определяется большая часть метаданных проекта. Используется для определения используемых моделей данных, отображений объектов и метаданных проектов;
- Operator – позволяет осуществлять пошаговое наблюдение за работой интерфейсов, пакетов программ, сценариев и загрузочных планов, а также используется при отладке;

- Topology Manager – используется для описания архитекуры информационной системы. Topology Manager работает только с главным репозиторием, с его помощью ODI имеет возможность использовать одни и те же интеграционные инферфейсы в различных физических средах;
- Security Manager – обеспечивает безопасность ODI. Используется для создания пользовательских профилей и призвоения им привилегий. Работает только с главным репозиторием.

Oracle GoldenGate поддерживает репликацию данных между неоднородными источниками данных. GoldenGate предлагает различные виды топологий, которые используют как технологии федеративных баз данных, так и технологии хранилищ данных:

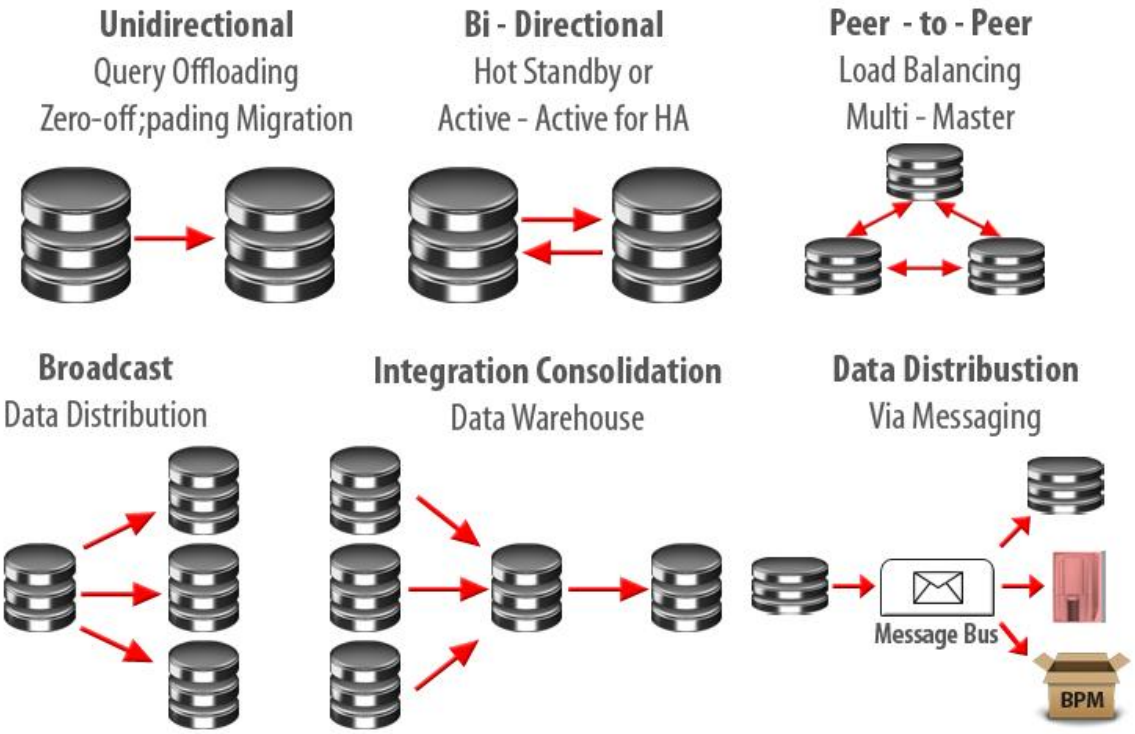


Рис. 1.2. Топологии Oracle GoldenGate

Архитектура Oracle GoldenGate состоит из следующих компонент:

- Manager – является главным процессом, запускающим остальные процессы GoldenGate. Данный процесс должен выполняться на исходной и целевой

системах с целью конфигурации и запуска остальных процессов GoldenGate, а также управления памятью.

- Extract – отвечает за сбор транзакций из логов Oracle Redo и последующую запись соответствующих изменений данных в Trail/Extract Files.
- Replicat – выполняется в конечном пункте цепи передачи данных на целевой базе данных. Данный процесс осуществляет изменения данных на целевых системах.
- Trail/Extract Files – отвечает за извлечение данных из источника и их передачу в удаленную систему либо в ее локальную репликацию;
- Collector – выполняется на целевой системе и осуществляет запись изменений данных источников в целевую систему.

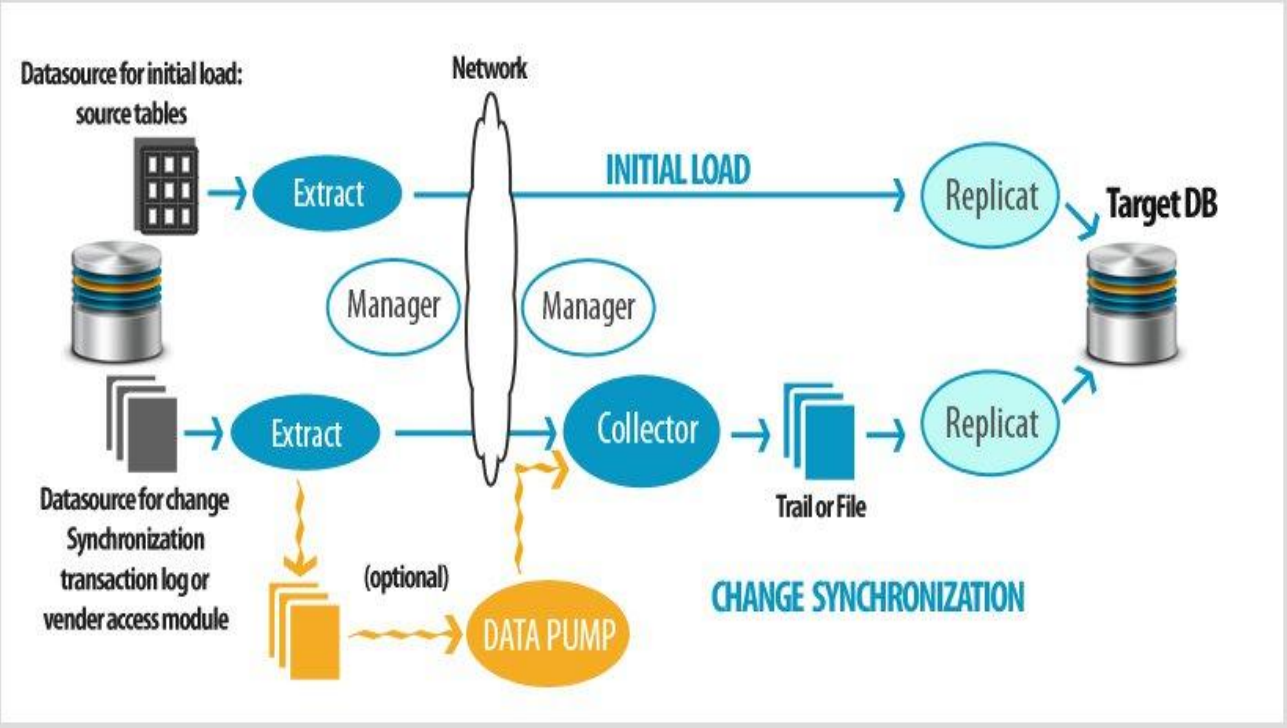


Рис. 1.3. Архитектура Oracle GoldenGate

Другим примером современных приложений, использующих технологию федеративных баз данных, является семейство программных продуктов в области управления информацией компании IBM - IBM DB2 [23]. DB2 была первой СУБД, в

которой использовался язык SQL. В основе DB2 лежит теория реляционных баз данных [24], [25], [26], разработанная и опубликованная Эдгаром Коддом в нале 1970-х гг.

Отличительными свойствами DB2 являются использование диалекта языка SQL, в котором, за редкими исключениями, определяется чисто декларативный смысл языковых конструкций, и использование многофазового оптимизатора, который строит эффективный план выполнения запроса с использованием данных декларативных конструкций. Также в диалекте SQL DB2 почти отсутствуют подсказки оптимизатору и мало развит язык хранимых процедур. Используются дополнительные механизмы манипулирования данными, например – табличные выражения, статистика распределения данных. В зависимости от статистических характеристик данных, один и тот же запрос может быть транслирован в различные планы выполнения.

1.2 Хранилища данных

Хранилища данных (data warehouses) – копии фрагментов информации из нескольких источников сохраняются в единой базе данных, возможно, с предварительной обработкой – фильтрацией (filtering), соединением (joining) или агрегированием (aggregating). Содержимое хранилища, как правило, обновляется на периодической основе. В процессе копирования данные подвергаются определенным преобразованиям с целью согласования их структур с общей схемой хранилища. Основным недостатком хранилищ данных является их потенциальное несоответствие реальному времени, так как изменения в источниках данных могут не быть своевременно в них отражены. Далее приведены описания некоторых современным систем, использующих технологию хранилищ данных.

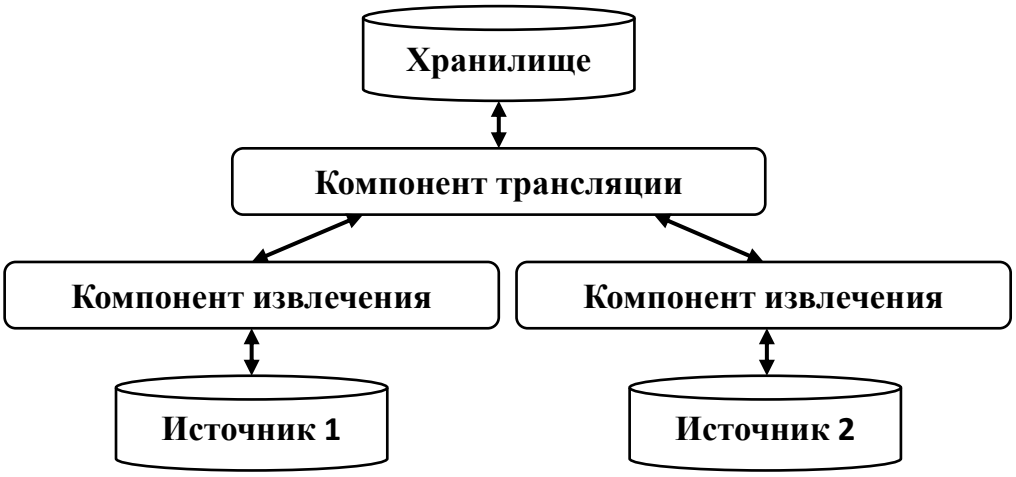


Рис. 1.4. Хранилище данных

SciDB [27], [28], [29], [30] является системой управления данными с открытым исходным кодом, нацеленная преимущественно на использование в областях, в которые вовлечены массивы данных очень больших (петабайты) размеров; например, научные приложения, такие как астрономия, дистанционное зондирование и моделирование климата, управление био-научной информацией, а также коммерческие приложения, такие как системы управления риском в услугах финансового сектора и анализы данных веб логов. SciDB использует модель данных, основанную на массивах (array data model). Базы данных SciDB организованы в виде коллекций n-мерных массивов. Каждая ячейка в SciDB массиве хранит кортеж значений, а отдельные значения в кортеже ассоциируются с различными именами атрибутов. Значения атрибутов SciDB могут принадлежать к любому из определенных числовых либо строковых типов фиксированной длины. Модель данных SciDB является *вложенной* (nested); ячейка массива SciDB может в свою очередь содержать другой SciDB массив.

При работе с данными базы данных SciDB пользователям предоставляется декларативный язык запросов. В основе данного языка запросов лежит небольшой набор алгебраических примитивов, оперирующих с массивами. Эти примитивы в общем случае могут быть охарактеризованы основываясь на том, работают ли они с массивами на уровне их структур – ранг массива, индексы измерений – либо на уровне

их контента – значений данных атрибутов в ячейках. Некоторые примитивы затрагивают оба уровня.

SciDB обычно устанавливается на сети компьютеров, на каждом из которых работает полуавтономный экземпляр машины SciDB, предоставляющий коммуникации, обработку запросов и управление локальной памятью. Процессы SciDB, работающие на каждом узле, имеют совместный доступ к (логически) централизованной базе системного каталога, которая хранит информацию об узлах, распределении данных, определенных пользователем расширениях и т.д. Данная архитектура создана под влиянием архитектур современных распределенных компьютерных систем, использующих модель Map/Reduce [31].

Архитектура системы хранения SciDB использует свойства некоторых коммерческих СУБД, но включает также некоторое количество новых особенностей, отражающих требования обработки массивов данных. SciDB использует распределенную систему хранения данных. Данные в массивах, аналогично системам, ориентированным на хранение столбцов (column-store), разделяются вертикально. Менеджер хранения SciDB расщепляет атрибуты в едином логическом массиве и обрабатывает значения каждого атрибута по отдельности. Иными словами, все операции низкого уровня в SciDB имеют дело с массивами, содержащими единственные значения в каждой ячейке. Пользователи-ученые часто фокусируют свое внимание на определенных подмножествах атрибутов массивов. Вертикальное разделение, таким образом, уменьшает стоимость ввода/вывода. Также, менеджер хранения SciDB, рассматривая данные каждого атрибута, разлагает соответствующие массивы на некоторое количество равных по размеру и потенциально пересекающихся сегментов (chunks). В SciDB сегменты являются физическими единицами ввода/вывода, обработки и межузлового сообщения. Сегменты имеют сравнительно большие размеры, порядка 64 МБ. В системном каталоге SciDB для каждого сегмента хранится его местонахождение в пределах логического массива. Пересечение сегментов позволяет распараллеливать некоторые операции. Отрицательной стороной данного подхода является увеличение затрат на хранение информации.

В последние годы получил широкую популярность ряд подходов к реализации хранилищ баз данных, относящихся к семейству NoSQL [32], [33], [34], которые ориентированы на обеспечение масштабируемости [35] и высокой производительности [36]. В отличие от традиционных систем, основанных на реляционной модели данных, системы NoSQL обычно поддерживают набор относительно простых операций, при этом обеспечивая гибкость структуры хранимых данных. Далее приводится описание некоторых современных NoSQL систем.

Couchbase [37], [38] является документно-ориентированной распределенной NoSQL базой данных для интерактивных приложений с открытым исходным кодом. Couchbase предоставляет легкую масштабируемость, стабильно высокую производительность и доступность 24/7. Сервер Couchbase хранит данные в виде JSON документов либо в бинарном формате. Документы равномерно распределяются по кластеру и хранятся в контейнерах данных, именуемых сегментами, с помощью которых осуществляется логическое группирование физических ресурсов в кластере. Сегменты разделяются на логические секторы, которые соответствуют определенным серверам кластера. Идентификаторы (ID) документов хешируются, и документы хранятся на серверах, соответствующим хешам их ID. Документы, хранимые на сервере Couchbase, могут быть проиндексированы. Вторичные индексы определяются с помощью проектных документов (design documents) и взглядов (views). Каждый проектный документ может иметь множество взглядов, и каждый сегмент Couchbase может иметь множество проектных документов. Индексы Couchbase являются распределенными, при этом на каждом сервере хранятся только индексы, касающиеся данных, которые на нем хранятся.

Родоначальником ориентированных на столбцы (column-oriented) баз данных является Big Table компании Google [39]. Модель данных основана на разреженной таблице, чьи строки могут содержать произвольные столбцы, ключи которых предоставляют естественное индексирование. Основными понятиями, используемыми для организации данных в данной модели, являются: столбец (column), состоящий из пары имя-значение; супер-столбец (super column), являющийся упорядоченным

множеством произвольного количества столбцов; столбцы хранятся в строках (raws), называемых также семействами столбцов (column families), а супер-столбцы хранятся в строках, известных как семейства супер-столбцов. Управление доступом и учет на уровне диска и памяти осуществляются на уровне семейств столбцов. Каждая ячейка BigTable может содержать множество версий одних и тех же данных, проиндексированное по времени. Различные значения ячейки хранятся в порядке убывания по времени, что позволяет более новым записям быть считанными в первую очередь.

Apache Cassandra [40] позволяет обрабатывать данные под большими нагрузками на многочисленных узлах без точек отказа. В Cassandra проблема отказа решается путем установления децентрализованной распределенной по однородным узлам системы. Узлы являются основными компонентами инфраструктуры в Cassandra. Данные распределяются по всем узлам кластера, индексируются и записываются в хранимую в памяти структуру, именуемую memtable. При ее заполнении, данные переносятся на диск в формате SSTable. Структурой индекса SSTable являются разреженные индексы, сопоставляющие ключам строк соответствующие отступы в файле данных.

Neo4j [41], [42] является представителем графовых баз данных. Данные представляются в виде узлов и их отношений, которые могут обладать свойствами. Узлы обычно используются для представления сущностей, хотя отношения тоже могут быть использованы с этой целью в зависимости от предметной области. Помимо отношений узлы могут также иметь некоторое количество меток. Запросы в Neo4j осуществляются с помощью декларативного языка Cypher для описания шаблонов на графах. Индексы создаются на основе комбинаций меток и свойств.

MongoDB [43] является документно-ориентированной базой данных с открытым исходным кодом, предоставляющей высокую производительность, доступность и легкую масштабируемость. База данных MongoDB состоит из коллекций, представляющих собой множества документов. Коллекция существует в рамках единой базы данных и не подразумевает наличия схемы. Каждый документ является

множеством пар ключ-значение и хранится в формате BSON (бинарно сериализованный JSON). Различные документы в коллекции могут иметь различные поля, хотя обычно документы одной коллекции имеют одинаковую или схожую структуру. MongoDB поддерживает различные типы индексов, а именно: по одному полю (single field), составные (compound), по нескольким полям (multikey), геопространственные (geospatial), текстовые и хешированные индексы. Индексы в MongoDB реализованы с помощью B-деревьев.

Использование эффективной структуры индекса в качестве адекватного формализма для управления данными является одной из основных задач при построении хранилища данных. Для ее решения в главе 3 диссертационной работы предложена динамическая структура индекса, основанная на концепции сеточных файлов [44], [45].

1.3 Медиаторы

Медиаторы (mediators) являются программными компонентами, которые обеспечивают поддержку виртуальных баз данных (virtual databases) [10]. Медиатор не сохраняет информацию сам по себе – вместо этого он транслирует запрос пользователя в один или несколько запросов, адресованных первичным источникам. После получения результатов обработки частных запросов, медиатор с их помощью синтезирует ответ на исходный запрос.

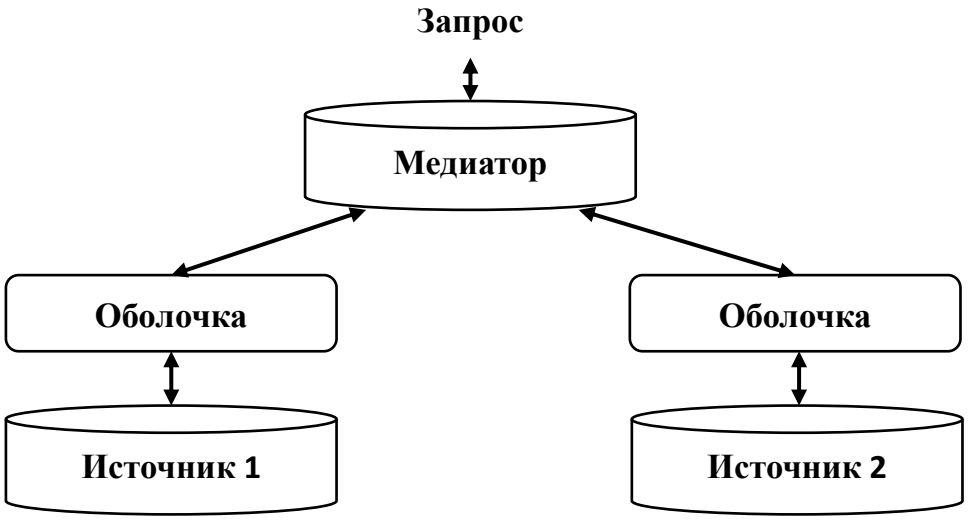


Рис. 1.5. Медиатор

Системы интеграции данных формально определяются в виде тройки $\langle G, S, M \rangle$, где G – глобальная схема (схема медиатора), S – множество схем источников, а M – отображение между запросами над глобальной схемой и схемами источников. В архитектурах основанных на идее медиатора, именуемых также архитектурами с посредником, используются три разновидности представления интегрированных данных – Global as View (GAV), Local as View (LAV) а также их совмещение, именуемое GLAV.

Система GAV [1], [2] подразумевает определение глобальной схемы G в терминах S. В этом случае M сопоставляет каждому элементу из G некоторый запрос над S, в результате чего обработка запросов становится относительно простой операцией. В этом случае вся сложность ложится на процесс добавления новых источников к системе, поскольку обновление схемы медиатора является длительным процессом. Поэтому подход GAV предпочтителен, когда источники информации не сильно изменяются в течение времени.

Напротив, в системах LAV [1], [2] источники информации представляются в терминах заданного интегрирующего глобального представления. В этом случае M сопоставляет каждому элементу из S некоторый запрос над G. При этом расширение G становится относительно легкой задачей, однако процесс обработки запроса в общем случае становится NP-полной задачей.

При GAV множество данных, представимых в источниках информации, может быть шире, чем возможно представить в медиаторе. Напротив, в LAV более ограничено множество данных, представимых в источниках, поэтому системы LAV в некоторых случаях могут быть не способны выполнить определенный запрос.

Подход GLAV [3] совмещает в себе преимущества LAV и GAV, предоставляя прямой доступ и источникам информации, в то же время сохраняя возможность определения глобальной схемы в терминах схем источников. Подход, изложенный в данной работе, следует отнести именно к категории GLAV.

В работах Паоло Атцени и др. [32], [46], [47], [48], [49] предложена система интеграции неоднородных NoSQL баз данных SOS (Save Our Systems). Целью данного подхода является создание однородного интерфейса, предоставляющего доступ к данным, хранимым в различных системах управления данными (в основном из семейства NoSQL, однако интеграция реляционных ресурсов также возможна), а также использования различных систем в рамках единого приложения. Архитектура системы SOS, представленная на рис. 1.6, организована с помощью следующих модулей:

1. Общий интерфейс, предлагающий примитивы для взаимодействия с NoSQL источниками данных;
2. Мета-уровень, хранящий информацию о вовлеченных данных;
3. Конкретные обработчики, генерирующие соответствующие вызовы к конкретным системам баз данных.

Интерфейс SOS представляет операции высокого уровня относительно общих конструкций, определенных на мета уровне. Мета уровень хранит данные и предоставляет интерфейс однородной модели данных для осуществления операций над объектами. Конкретные обработчики поддерживают взаимодействие с конкретными NoSQL системами хранения путем отображения общих вызовов мета уровня в запросы, специфичные данным системам.

Мета уровень спроектирован для эффективной работы с различными NoSQL моделями данных, позволяя поддерживать коллекции объектов, имеющих произвольную вложенную структуру. Тремя основными видами конструкций данной

модели являются структуры, множества и атрибуты. Каждый экземпляр базы данных представляется в виде множества коллекций. Каждая коллекция, в свою очередь, является множеством, содержащим произвольное количество объектов. Каждый объект идентифицируется уникальным для соответствующей коллекции ключом. Простые элементы, такие как пары ключ-значение либо отдельные спецификаторы, могут быть представлены как атрибуты. Группы атрибутов, такие как семейства столбцов Hbase либо хеши Redis, представляются в виде структур. Наконец, коллекции элементов, такие как таблицы HBase либо коллекции MongoDB, моделируются посредством множеств. Согласно конкретным структурам, реализованным в различных NoSQL системах, определяется процесс трансляции конструкций мета уровня в когерентные, специфичные данным системам, структуры.

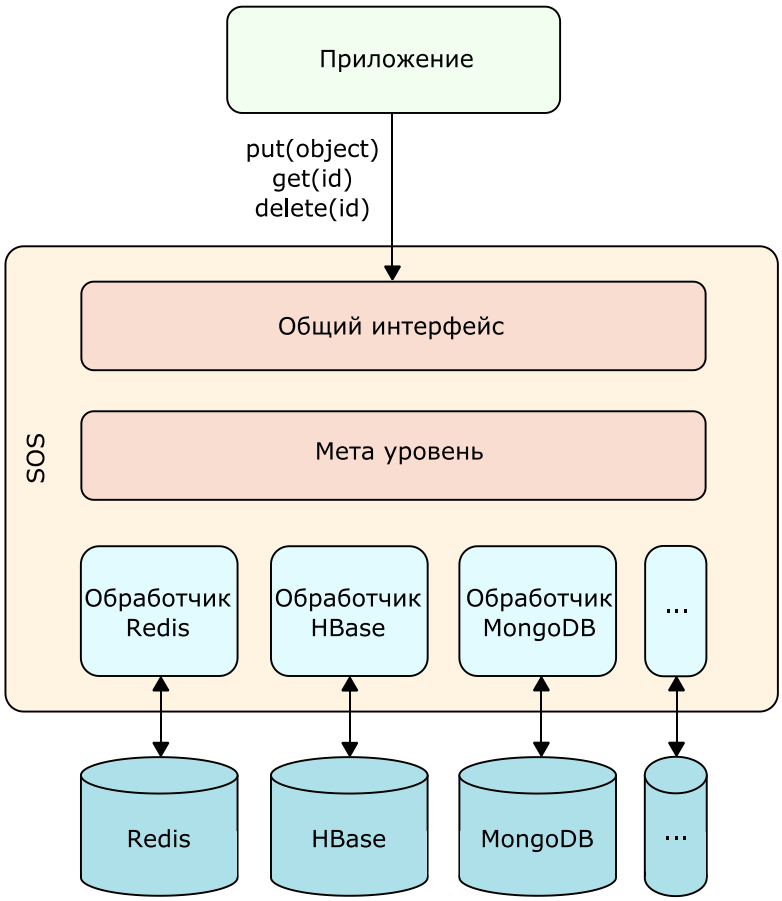


Рис. 1.6. Архитектура SOS

Был разработан ряд интеграционных систем, в основе которых лежит модель данных XML [50], [51], [52], [53], [54], [55], и используется язык запросов XQuery [56] (изначально – XML-QL [57]) [19]. В проекте Tsimmis [58] впервые были проиллюстрированы преимущества использования полуструктурированной модели данных при интеграции данных.

Необходимость поддержки интероперабельности неоднородных систем на основе отображений моделей и схем была указана Бернштейном и Мельником [59]. Некоторые предложения в этом направлении для традиционных (реляционной и объектной) моделей были сформулированы Тервиллигером и др. [60] и Морком и др. [61]. С методологической точки зрения, необходимость однородной классификации и обобщения принципов для NoSQL баз данных находит всеобщее признание; в работе Каттелла [62] была представлена детальная характеристика нереляционных систем. Стоунбрейкер [63] приводит аргументы в пользу ослабления важности NoSQL систем в научном контексте. В частности, Стоунбрейкер денонсирует отсутствие консолидированного стандарта для NoSQL моделей. Также он использует отсутствие формального языка запросов в качестве дополнительного аргумента.

1.3.2 Принципы синтеза канонической модели

В теоретической основе предлагаемого в данной работе подхода к интеграции данных лежат работы группы SYNTHESIS (ИПИ РАН) [64], [65], [66], [67], [68], [7] которые являются пионерами в области исследования методов интеграции неоднородных источников информации. Можно выделить три этапа развития исследований в этой области:

1. Период структурированных моделей данных (доминировавший до середины 80х годов XX века);
2. Период объектных моделей и интероперабельного конструирования систем (начался в конце 80х годов);

3. Период поведенческих, процессных моделей (начался в середине 90х годов).

Группой SYNTHESIS был впервые предложен метод синтеза канонической модели и были сформулированы принципы, обеспечивающие эквивалентность моделей данных. В основе этих принципов лежит идея расширения канонической модели данных таким образом, чтобы стало возможным отображение информации и операторов в нее из моделей данных источников. Помимо этого, все операции над схемами и моделями данных производятся на уровне некоторой метамодели данных – формального языка, обладающего достаточной мощью, чтобы иметь возможность представлять всевозможные концепции различных моделей данных. В качестве такой метамодели первоначально использовалась денотационная семантика, однако в скором времени была замещена нотацией абстрактных машин [12], [69]. В рамках этой нотации понятие эквивалентности моделей данных сводится к понятию уточнения соответствующих спецификаций. Основным преимуществом использования AMN является B-технология [12], [70], [71], [72], предоставляющая возможность полуавтоматического доказательства факта уточнения. Наличие подобной возможности доказательства существования отображений из исходных моделей данных в целевую является причиной выбора подхода группы SYNTHESIS в качестве основы для предлагаемого в диссертационной работе подхода к интеграции данных.

В работах группы SYNTHESIS были впервые определены понятия эквивалентности состояний баз данных, схем и моделей данных и описаны принципы сохранения операций и информации в процессе конструирования отображений неоднородных источников информации в каноническую модель. Согласно этому подходу, в рамках СУБД каждая модель данных определяется синтаксисом и семантикой двух языков – языка определения данных (ЯОД) и языка манипулирования данными (ЯМД). Основными принципами отображения произвольной исходной модели данных в целевую являются [7]:

1. *Принцип аксиоматического расширения моделей данных.* Согласно этому принципу, каноническая модель должна быть расширяемой. При этом ее расширение при рассмотрении каждой новой модели данных носит аксиоматический характер:

целевая модель данных расширяется путем добавления к ее ЯОД набора аксиом, определяющих логические зависимости данных исходной модели в терминах целевой. Полученное расширение должно быть эквивалентно исходной модели

2. *Принцип коммутативного отображения моделей данных*. Согласно этому принципу, сохранение операций и информации исходной модели при ее отображении в каноническую достигается при условии коммутативности диаграммы отображения ЯОД (схем) и ЯМД (операций).

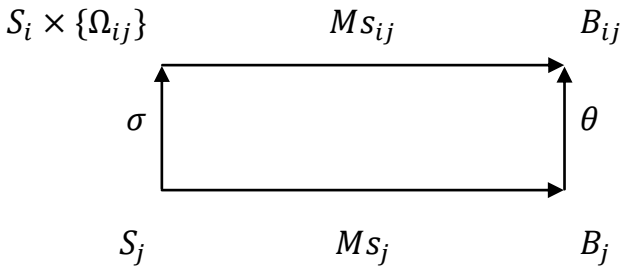
Множество всех схем, выразимых в ЯОД модели данных M_i, обозначается S_i, а множество операторов ЯМД модели M_i обозначается O_i. *Пространство допустимых состояний*, выразимых в M_i, обозначается B_i. Тогда

$Ms_i : S_i \to B_i$ есть семантическая функция ЯОД M_i.

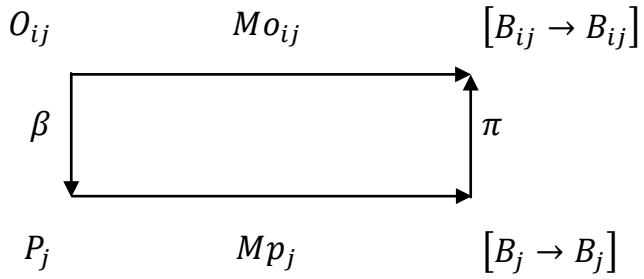
$Mo_i : O_i \to [B_i \to B_i]$ есть семантическая функция ЯМД M_i.

При этих обозначениях, отображение $f = \langle \sigma, \theta, \beta \rangle$ модели M_j в расширение M_{ij} модели M_i коммутативно, если выполняются следующие условия:

- Диаграмма отображения схем является коммутативной:



- Диаграмма отображения операторов является коммутативной:



- Отображение θ биективно.

Здесь Ω_{ij} обозначает множество схем аксиом, выражающих зависимости данных модели M_j в терминах модели M_i, а P_j обозначает последовательность операторов ЯМД модели M_j.

3. *Принцип синтеза унифицирующей канонической модели данных*. Синтезом канонической модели называется процесс построения расширений его ядра, эквивалентных различным моделям данных, включаемых в среду, а также процесс слияния этих расширений с канонической моделью. Согласно этому принципу, в создаваемой унифицирующей канонической модели разнообразные исходные модели данных имеют однородное эквивалентное представление.

Таким образом, можно выделить следующие основные задачи, возникающие при интеграции данных:

- Построение расширяемой канонической модели данных;
- Определение моделей данных с помощью формальной метамодели;
- Построение отображений из исходных моделей данных в каноническую;
- Доказательство корректности построенных отображений.

Для решения данных задач в главе 2 диссертационной работы предлагается расширяемая каноническая модель данных. В качестве формальной метамодели данных используется нотация абстрактных машин, описание которой приведено ниже.

1.3.2 Нотация абстрактных машин

Нотация абстрактных машин была впервые использована как формальная метамодель данных в начале 90-х годов [70]. AMN обеспечивает манипулирование теоретико-множественными спецификациями в логике первого порядка и доказательство уточнения спецификаций. С помощью техники уточнения стало возможным расширить основные определения отношений между типами данных, схемами и моделями данных так, чтобы вместо эквивалентности соответствующих спецификаций можно было рассуждать об их уточнении [73]. Специальные инструментальные средства (B-технология) представляют возможность доказательства

коммутативности диаграмм отображения моделей полуавтоматическим способом: теоремы, требуемые для доказательства уточнения моделей, генерируются B автоматически, но их доказательство может требовать вмешательства человека [74].

AMN как абстрактная метамодель

AMN, как теоретико-модельная нотация, позволяет рассматривать интегрированно спецификацию пространства состояний и поведения (определенного операциями на состояниях). Состояние машины определяется значениями ее переменных, а множество допустимых состояний задается с помощью инварианта – ограничения, которое должно всегда удовлетворяться.

Ключевым понятием AMN является *уточнение*, которое позволяет сопоставлять спецификации систем различных уровней абстракции. Уточняющая спецификация может быть намного более детальной, чем уточняемая спецификация. Уточняющая спецификация строится на основе алгоритмического уточнения и уточнения данных. Формально понятие уточнения определяется в AMN с помощью множества теорем специального вида, именуемых proof obligations. С использованием специальных инструментальных средств поддержки B-технологии, данные теоремы могут быть сформулированы автоматически. Их доказательство может быть произведено полуавтоматически – в некоторых случаях процесс доказательства может требовать вмешательства человека.

Спецификация состояний системы в AMN

Язык спецификации состояний основан на теории множеств Цермело-Френкеля с аксиомой выбора и типизированном языке первого порядка со встроенными типами и конструкторами сложных типов [7]. Примерами конструкторов сложных типов являются:

- $(s \times t)$ – декартово произведение множеств s и t;
- $(\mathbb{P}(s))$ – множество всех подмножеств множества s;

- $(\{x|x \in s \wedge P(x)\})$ – выделение из множества s подмножества элементов, удовлетворяющих предикату P;
- $(s \cap t)$ – пересечение множеств s и t;
- $(s \cup t)$ – объединение множеств s и t;
- $(s - t)$ – разность множеств s и t;
- конструкторы реляционных сортов $(s \leftrightarrow t)$,
- конструкторы функциональных сортов $(s \rightarrow t)$.
- конструкторы упорядоченной пары обозначается как (x, y) либо как $(x \mapsto y)$.

Отношение между множествами s и t представляет собой элемент множества $\mathbb{P}(s \times t)$. Область определения отношения r обозначается как $dom(r)$, а область допустимых значений как $ran(r)$. Функция f из множества s в множество t представляется в виде отношения между данными множествами, при этом каждому элементу $dom(f) \subseteq s$ сопоставлен ровно один элемент множества t. Существуют обозначения для частичных функций $f \in s \nrightarrow t$ и полных функций $f \in s \rightarrow t$. Для функции f и элемента $x \in dom(f)$ определены терм $f(x)$, и его значение y $(y = f(x))$ такое, что $x \longmapsto y \in f$.

В таблице 1.1 представлены основные теоретико-множественные обозначения AMN. Здесь S, T, U, s, t являются множествами, такими что $s \in S$ и $t \in T$, r, r_1, r_2, p, q являются отношениями из S в T, а q является отношением из T в U.

Язык предикатов AMN сочетает теорию множеств и исчисление предикатов первого порядка (с обычным набором связок $\wedge, \vee, \Longrightarrow, \neg$, кванторами существования $\exists$ и всеобщности $\forall$ и предикатом принадлежности $\in$) с простой арифметической теорией. Интерпретация состояния абстрактной машины осуществляется переменными машины – каждой переменной присваивается элемент из определенного домена.

Таблица 1.1. Теоретико-множественные обозначения AMN

Обозначение AMN	Значение обозначения
$S \leftrightarrow T$	$\mathbb{P}(S \times T)$
$\mathrm{dom}\,(r)$	$\{x \mid x \in S \wedge \exists y \bullet (y \in T \wedge x \mapsto y \in r)\}$
$\mathrm{ran}\,(r)$	$\{y \mid y \in T \wedge \exists x \bullet (x \in S \wedge x \mapsto y \in r)\}$
$p;\ q$	$\{x \mapsto z \mid x \mapsto z \in S \leftrightarrow U \wedge$ $\exists y \bullet (y \in T \wedge x \mapsto y \in p \wedge y \mapsto z \in q)\}$
$\mathrm{id}\,(S)$	$\{x \mapsto y \mid x \mapsto y \in S \times S \wedge x = y\}$
$s \lhd r$	$\{x \mapsto y \mid x \mapsto y \in r \wedge x \in s\}$
$s \rhd r$	$\{x \mapsto y \mid x \mapsto y \in r \wedge y \in t\}$
$s \lhd\!\!\!- r$	$\{x \mapsto y \mid x \mapsto y \in r \wedge x \in S - s\}$
$s \rhd\!\!\!- r$	$\{x \mapsto y \mid x \mapsto y \in r \wedge y \in T - t\}$
r^{-1}	$\{y \mapsto x \mid y \mapsto x \in T \times S \wedge x \mapsto y \in r\}$
$r[s]$	$\{y \mid y \in T \wedge \exists x \bullet (x \in s \wedge x \mapsto y \in r)\}$
$r_1 \lhd\!\!\!- r_2$	$(\mathrm{dom}\,(r_2) \lhd\!\!\!- r_1) \cup r_2$
$s \nrightarrow t$	$\{r \mid r \in s \leftrightarrow t \wedge (r^{-1};\ r) \subseteq \mathrm{id}\,(t)\}$

Спецификация операций AMN

Операции абстрактных машин описываются при помощи языка обобщенных подстановок (Generalized Substitution Language, GSL) и имеют следующий вид:

$$r_1, \ldots, r_n \longleftarrow op(p_1, \ldots, p_m) = S$$

Здесь op – имя операции, $r_1, \ldots, r_n$ – выходные параметры операции, $p_1, \ldots, p_m$ – входные параметры операции, S – подстановка, определяющая действие операции на пространстве состояний.

GSL позволяет описывать переходы между состояниями системы [7]. Каждая обобщенная подстановка S определяет преобразователь предиката, связывающий некоторое постусловие R со своим слабейшим предусловием $[S]R$, что гарантирует сохранение R после выполнения операции. В таком случае говорят, что S устанавливает R. "Слабейшее" предусловие означает, что предикат "начального состояния", связанный с некоторым предикатом "заключительного состояния", должен разрешать максимально большое число состояний. В таблице 1.2. приведены основные виды обобщенных подстановок и слабейшие предусловия, которые им соответствуют. Здесь S, T, T_1, T_2, S_1, S_2 являются подстановками, x, y, t - переменными, E, F - выражениями, G, G_1, G_2, H, P – предикатами. $P\{x \to e\}$ есть предикат P, в котором все свободные вхождения переменной x заменены на E.

Таблица 1.2. Обобщенные подстановки и их семантика

Обобщенная подстановка S	$[S]P$
$x := E$	$P\{x \to E\}$
skip	P
$x := E \mid\mid y := F$	$[x, y := E, F]P$
$S[]T$	$[S]P \wedge [T]P$
SELECT G_1 THEN T_1 WHEN G_2 THEN T_2 END	$(G_1 \Rightarrow [T_1]P) \wedge (G_2 \Rightarrow [T_2]P)$
PRE G THEN T END	$G \wedge [T]P$
ANY t WHERE G THEN T END	$\forall t \bullet (G \Rightarrow [T]P)$
S ; T	$[S][T]P$
IF G THEN S ELSE T END	$(G \Rightarrow [S]P) \wedge (\neg G \Rightarrow [T]P)$

Виды конструкций и структурные механизмы AMN

В AMN существует три вида конструкций:

- абстрактная машина (abstract machine),
- уточнение (refinement),
- реализация (implementation).

Абстрактная машина может быть использована только как уточняемая конструкция, при этом в ее описании не должны присутствовать последовательные и циклические подстановки. Реализация может быть использована только как уточняющая конструкция, при этом в ее описании не должны присутствовать недетерминированные подстановки (SELECT, ANY), параллельные подстановки и предусловия. Реализация также не должна иметь собственных переменных. Уточнение является более универсальной конструкцией, т.к. оно может быть использовано и в качестве уточняемой, и в качестве уточняющей конструкции. Ограничений на описание операций, подобных ограничениям для абстрактной машины и реализации, для уточнения не существует. По этой причине уточнение является наиболее предпочтительной конструкцией для однородного представления спецификаций канонической модели в AMN.

Конструкция уточнения описывается следующим образом:

REFINEMENT r
REFINES m
SEES sm
INCLUDES im
SETS s
CONSTANTS c
PROPERTIES $P(s, c)$
VARIABLES x
INVARIANT $I(x)$
INITIALISATION S
OPERATIONS $O_1, \dots, O_n$
END

При помощи переменных, описанных в разделе *VARIABLES*, определяется состояние системы. Инициализация переменных определяется подстановкой, описанной в разделе *INITIALISATION*. Состояние системы может быть изменено при помощи операций, определенных в разделе *OPERATIONS*. После инициализации, а также при выполнении каждой операции, состояние системы должно удовлетворять инварианту, описанному в разделе *INVARIANT*. В разделе *SETS* содержатся определения множеств, в разделе *CONSTANTS* – наименования констант, которые используются в уточнении, в разделе *PROPERTIES* – предикат, с помощью которого описываются свойства констант. С помощью разделов *SEES* и *INCLUDES* возможно осуществление композиции уточнения с другими конструкциями. Композиция *SEES* используется в том случае, когда некоторому множеству конструкций необходимо дать возможность чтения значений переменных, множеств и констант некоторой спецификации. Композиция *INCLUDES* используется в том случае, когда конструкции необходимо включить некоторую конструкцию в качестве своей подсистемы. При этом переменные включаемой конструкции становятся переменными включающей конструкции, однако изменение их значений может быть осуществлено только с помощью операций включаемой конструкции. Инвариант включаемой конструкции становится частью инварианта включающей конструкции.

Формализация понятия уточнения в AMN

Рассмотрим метод формализации факта уточнения конструкции M конструкцией N в AMN [7]. Факт уточнения может быть установлен только в том случае, если конструкции M и N (таблица 1.3) согласованы, т.е. удовлетворяют следующим требованиям:

- Раздел REFINES конструкции N должен содержать имя M.
- Для каждой операции конструкции M, конструкция N должна содержать операцию с точно такой же сигнатурой.
- Инвариант конструкции N должен содержать так называемый склеивающий инвариант (инвариант уточнения) R, задающий соотношение между состояниями уточняемой и уточняющей конструкций.

Таблица 1.3. Спецификации M и N

MACHINE M CONSTANTS C_M PROPERTIES P_M VARIABLES v INITIALISATION $Init_M$ INVARIANT I_M OPERATIONS $y \leftarrow op(x) =$ PRE $Pre_{op,M}$ THEN $Def_{op,M}$ END … END	REFINEMENT N REFINES M CONSTANTS C_N PROPERTIES P_N VARIABLES w INITIALISATION $Init_N$ INVARIANT I_N OPERATIONS $y \leftarrow op(x) =$ PRE $Pre_{op,N}$ THEN $Def_{op,N}$ END … END

Определение 1.3.1. N уточняет M, если верны следующие теоремы (proof obligations).

1. Теорема непустоты объединенного состояния. Существует объединенное состояние M и N, удовлетворяющее инвариантам M и N.

$$P_M \wedge P_N \Longrightarrow \exists(v, w) \cdot (I_M \wedge I_N)$$

2. Теорема уточнения инициализации. Инициализация N уточняет инициализацию M.

$$P_M \wedge P_N \Longrightarrow [Init_N]\neg[Init_M]\neg I_N$$

3. Теорема уточнения операций. Каждая из операций N уточняет соответствующую операцию M, т.е. при условии выполнения инварианта уточнения и предусловия уточняемой операции, выполняется предусловие уточняющей операции; и для каждого случая исполнения $Def_{op,N}$, существует исполнение $Def_{op,M}$ из соответствующего начального состояния (задаваемого инвариантом уточнения), которое устанавливает точно такие же значения на пост-состояниях.

$$P_M \wedge P_N \wedge I_M \wedge I_N \wedge P_M \wedge Pre_{op,M} \Longrightarrow$$
$$Pre_{op,N} \wedge \left[Def_{op,N}\{y \longrightarrow y'\}\right]\neg\left[Def_{op,M}\right]\neg(I_N \wedge y' = y)$$

1.4 Выводы по главе

В главе 1 был предложен анализ основных подходов к интеграции данных.

- Сформулированы задачи, возникающие при виртуальной и материализованной интеграции данных;
- Изложены основные принципы отображения моделей данных, предложенные группой SYNTHESIS;
- Приведены доводы в пользу выбора подхода группы SYNTHESIS в качестве основы для предлагаемого в данной работе подхода к интеграции данных;
- Описана нотация абстрактных машин, используемая в качестве метамодели данных.

2 XML-КАНОНИЧЕСКАЯ МОДЕЛЬ ДАННЫХ

В рамках предлагаемого подхода к интеграции данных создана расширяемая каноническая модель, основанная на XML. Предлагается принцип расширения ядра канонической модели. В целях создания обоснованного отображения для интеграции неоднородных баз данных, концепция модели данных формализуется посредством нотации абстрактных машин. Для каждой исходной модели создается обратимое отображение в расширение канонической модели. B-технология используется для доказательства того, что AMN-семантика исходной модели представляет собой уточнение AMN-семантики расширенной канонической модели. Таким образом доказывается корректность отображения и возможность использования расширенной канонической модели для представления схем исходной модели. Созданы AMN-машины для канонической и реляционной моделей, а также для реляционного уточнения канонической модели.

2.1 Модель данных XDM

В качестве ядра канонической модели данных для медиатора была выбрана модель данных XDM [75], [76]. Выбор обусловлен рядом преимуществ этой модели данных, в первую очередь ее гибкостью. Она представляет из себя компромисс между структурированными (реляционная, объектная) и полуструктурированными (XML) моделями данных.

XDM появилась в результате расширения модели данных XML в целях поддержки концепции базы данных. В рамках этой модели рассматриваются базовые объекты, такие как целые числа, строки, переменные различных типов, символы, и составные объекты, определенные в терминах λ-исчисления [77]. Для получения сложных типов используются следующие конструкции:

1. *Attribution*. Если v переменная, а t – типизированный объект, то **attribution**$(v, type\ t)$ есть типизированый объект, обозначающий переменную типа t.

2. *Abstraction*. Если v переменная, а t, A – типизированные объекты, то **binding**$(lambda$, **attribution**$(v, type\ t)$, $A)$ есть типизированный объект.

3. *Application*. Если F и A – типизированные объекты, то **application**(F, A) есть типизировнный объект.

4. *Function Space*. Если t и u – типизированные объекты, и v – переменная, то **binding**$(PiType$, **attribution**$(v, type\ t)$, $u)$ есть типизированный объект. Он представлет тип функций, отображающих аргумент v типа t в результат типа u.

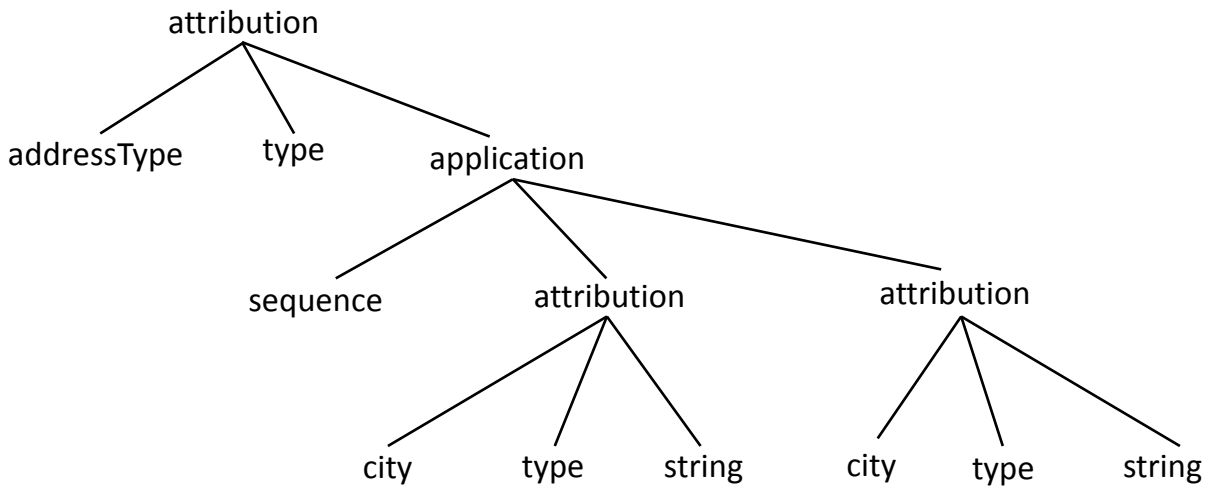


Рис. 2.1. Представление сложного типа addressType посредством конструкций XDM

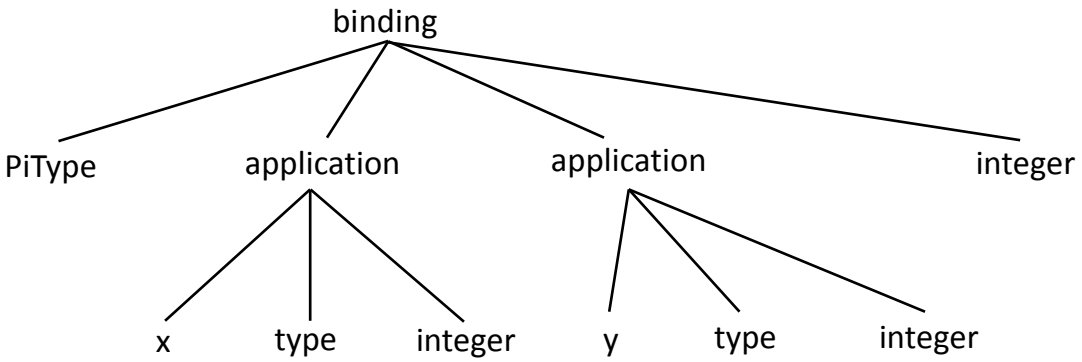


Рис. 2.2. Формализация сигнатуры функции с помощью конструкции binding

Основными понятиями алгебраической модели XDM являются понятия схемы (xdm-schema) и элемента (xdm-element) [75]:

Определение 2.1.1. S называется схемой, если $S = \langle name,\ type,\ f \rangle$ или $S = \langle name,\ typeOp(S_1, S_2, \dots, S_n),\ f \rangle$, и S_i – схема, где $typeOp \in \{sequence, choice, all\}$, $f \in \{?, *, +, \perp\}$.

Определение 2.1.2. Элементом s схемы S называется конечное множество отображений

$S \to domain\big(firstComp(S)\big) \times domain\big(secondComp(S)\big)$;

Если $secondComp(S) = typeOp(S_1, S_2, \dots, S_n)$, то должно выполняться ограничение $\forall e \in s \cdot \big(\ e[S_i] \in domain(S_i)\big),\ 1 \le i \le n.$

В роли ЯМД модели данных XDM выступает алгебра элементов (element algebra) [78], [79]. Семантика ее операций определена аналогично семантике операций реляционной алгебры. Ниже приводятся определение подобия схем, а также некоторых алгебраических операций.

Определение 2.1.3. Пусть R и Q схемы. Скажем что R и Q подобны, если $secondComp(R) = atomicType1$, $secondComp(Q) = atomicType2$ и $atomicType1 = atomicType2$

или

$secondComp(R) = typeOp(R_1, \dots, R_n), secondComp(Q) = typeOp(Q_1, \dots, Q_n)$ и R_i, Q_i подобны для всех $1 \le i \le n$.

Конкатенация элементов $r =< name_r, r_1, \dots, r_k >$ и $q =< name_q, q_1, \dots, q_m >$ определяется как $\widehat{rq} =< \widehat{rq}, r_1, \dots, r_k, q_1, \dots, q_m >$

Теоретико-множественные операции. Пусть r и q элементы с подобными схемами R и Q. Операции объединения, пересечения и разности этих элементов определяются следующим образом:

$$r \cup q =< r \cup q, \langle t | t \in r \vee t \in q \rangle >$$
$$r \cap q =< r \cap q, \langle t | t \in r \wedge t \in q \rangle >$$
$$r - q =< r - q, \langle t | t \in r \wedge t \notin q \rangle >$$

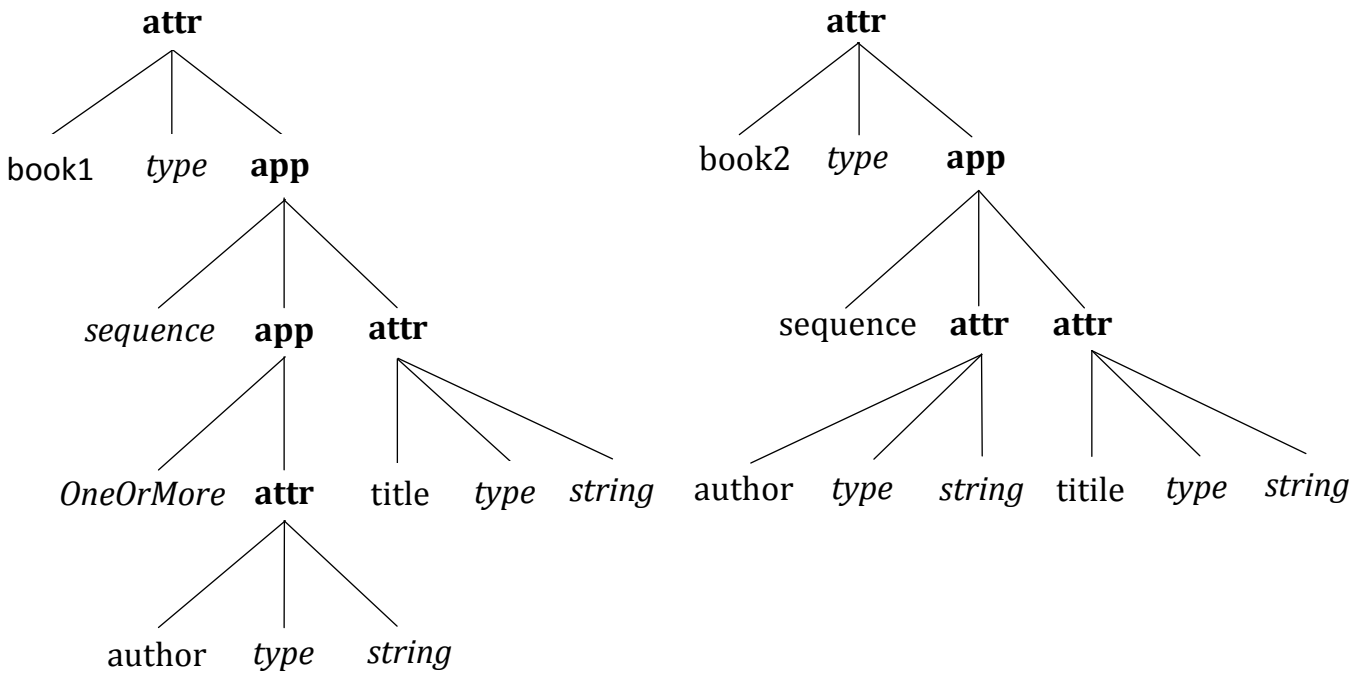


Рис 2.3. Схемы book1 и book2 подобны

Декартово произведение. Пусть r и q элементы со схемами R и Q соответственно. Декартовым произведением r и q является элемент

$$r \times q =< r \times q, \langle \widehat{ts} | t \in r \wedge s \in q \rangle >$$

Выбор. Пусть r есть элемент со схемой R, а P – предикат. Результатом выбора из r с помощью P является элемент

$$\sigma_P(r) =< \sigma_P(r), \langle t | t \in r \wedge P(t) \rangle >$$

Удаление дупликатов. Пусть r есть элемент со схемой R. Результатом удаления дупликатов из r является элемент

$$\delta(r) =< \delta(r), \langle t | t \in r \rangle >$$

2.2 Формализация концепции интеграции данных с помощью XML

В данном параграфе предлагается подход к интеграции данных, основанный на онтологии. В рамках данного подхода рассматривается как виртуальная, так и материализованная интеграция данных в пределах канонической модели. Таким образом, возникает необходимость формализации понятий данной предметной области, таких как медиатор, хранилище данных и схема базы данных. В данном случае онтология интеграции данных основана на XML-формализации этих понятий, которая определяется с помощью следующей DTD:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT DIO (DBSCH | MED | WHSE)*>
<!ELEMENT DBSCH (OMATTR)+>
<!ELEMENT MED (MSCH, WRAPPER, CONSTRAINT*)>
<!ELEMENT MSCH (OMATTR)>
<!ELEMENT WRAPPER (OMOBJ)>
<!ELEMENT CONSTRAINT (OMOBJ)>
<!ATTLIST MED name CDATA #REQUIRED>
<!ELEMENT WHSE (WSCH, EXTRACTOR, GRID)>
<!ELEMENT WSCH (OMATTR)>
<!ELEMENT EXTRACTOR (OMOBJ)>
<!ATTLIST WHSE name CDATA #REQUIRED>
<!ELEMENT GRID (DIM+, CHUNK+)>
<!ELEMENT HASH (OMOBJ)>
<!ELEMENT DIM (STRIPE)+>
<!ELEMENT STRIPE EMPTY>
<!ELEMENT CHUNK (AVG)+>
<!ELEMENT AVG EMPTY>
<!ATTLIST DIM name CDATA #REQUIRED>
<!ATTLIST AVG    value CDATA #IMPLIED
            dim  CDATA #REQUIRED>
<!ATTLIST CHUNK  id ID #REQUIRED
```

```
        qty CDATA #REQUIRED
        ref_to_db CDATA #REQUIRED
        ref_to_chunk IDREFS #IMPLIED>
<!ATTLIST STRIPE ref_to_chunk IDREFS #IMPLIED
        min_val CDATA #REQUIRED
        max_val CDATA #REQUIRED
        rec_cnt CDATA "0"
        chunk_cnt CDATA "0">
```

Ядро канонической модели реализуется как XML-приложение. Его синтаксис определяется синтаксическими правилами XML, его грамматика частично определяется его собственной DTD. На уровне DTD может быть обеспечена только синтаксическая достоверность объектов ядра. Для проверки семантики, в дополнение к общим правилам, унаследованным XML-приложениями, рассматриваемое приложение определяет новые синтаксические правила. Это достигается путем представления *словарей контента* (content dictionaries). Словари контента используются для присвоения формальной и неформальной семантик всем символам (например, type, sequence, OneOrMore, string, и т.д.), используемым в объектах ядра. Словарь контента является коллекцией связанных элементов, представленных в формате XML. Иными словами, каждый словарь контента определяет символы, представляющие понятия из некоторой специфичной предметной области.

Понятия медиатора, хранилища данных и схемы базы данных представляются с помощью XML-элементов *MED*, *WHSE* и *DBSCH* соответственно. Содержимое элемента *dbsch* основано на элементе ядра *OMATTR* [80], посредством которого возможно моделирование схем баз данных. Содержимое элемента *MED* основано на элементах *MSCH, WRAPPER, CONSTRAINT* и имеет атрибут *name* – наименование медиатора. Элемент *MSCH* интерпретируется аналогично элементу *DBSCH,* только используется при моделировании схем медиатора. Содержимое элементов *WRAPPER* и *CONSTRAINT* базируется на элементе ядра *OMOBJ* (математический объект, подробнее см. [80]). Посредством элемента *WRAPPER* определяются отображения из исходных моделей

данных в каноническую. Ограничения целостности на уровне медиатора определяются значениями элементов *CONSTRAINT*. Очень важно, что для определения отображений и ограничений целостности используется вычислительно полный язык [80], [81]. Предлагаемый подход к поддержке хранилищ данных основывается на понятии сеточных файлов, и формализуется посредством элемента *WHSE*, содержащего элементы *WSCH, EXTRACTOR* и *GRID* и имеющего атрибут *name*. Элементы *WSCH* и *EXTRACTOR* для хранилища аналогичны элементам *MSCH* и *WRAPPER* для медиатора. Элемент *GRID* используется для формализации концепции сеточного файла, и подробно описывается в §3.3.

2.3 Принцип расширения ядра канонической модели

Расширение ядра канонической модели производится при рассмотрении моделей данных новых источников информации путем добавления новых символов (понятий) к ее ЯОД в целях выражения логических зависимостей исходной модели средствами целевой. Полученное расширение должно стать эквивалентным исходной модели данных. Для реализации расширения канонической модели используется следующее правило:

Concept ← *Symbol* ContextDefinition

Например, для поддержания таких понятий реляционной модели как ключ и ссылочная целостность, ядро канонической модели дополняется следующими символами: *key, foreign key, unique, constraint, on update, on delete, cascade, set null*. В общем случае расширение ядра сводится к добавлению новых символов в словари контента, представленныи в виде XML-документов, и определению контекстов применения данных символов. Словари контента используются для присвоения формальных и неформальных семантик всем понятиям, используемым в моделях данных

Рассмотрим для примера следующие схемы отношений:

S = {Snum, Sname, Status, City}

P = {Pnum, Pname, Color, Weight, City}

SP = {Snum, Pnum, Qty}

В расширении канонической модели, с использованием добавленных символов, данные схемы будут представлены в следующем виде:

S ← *attribution*(S, *type* TypeContext, *constraint* ConstContext)

TypeContext ← *application*(*sequence*, ApplicationContext)

ApplicationContext ← *attribution*(Snum, *type int*), *attribution*(Sname, type string), attribution(Status, *type int*), attribution(City, *type string*)

ConstraintContext ← *attribution*(ConstraintName, *key* Snum)

SP ← *attribution*(SP, *type* TypeContext, *constraint* KeyConstraintContext, *constraint* ForeignKeyConstraintContext1, *constraint* ForeignKeyConstraintContext2)

KeyConstraintContext ← *attribution*(ConstraintName, *key application*(*list*, Snum, Pnum))

ForeignKeyConstraintContext ← *attribution*(ConstraintName, *foreign key application*(*ref*, ConstraintNameOFS), *on update cascade, on delete cascade*)

Существенно, что для определения контекста используется вычислительно полный язык [80], [81]. В результате данного подхода, добавление новых символов в ЯОД не приводит к изменению парсера ЯОД.

2.4 Формализация моделей данных

В рамках предлагаемого подхода к интеграции данных, модели данных и отображения между ними рассматриваются как экземпляры метамодели. В качестве метамодели данных используется нотация абстрактных машин. Иными словами, AMN-формализм используется для определения концепций моделей данных. Каждая используемая модель данных формализуется в AMN в виде некоторой абстрактной машины (либо иерархии абстрактных машин), представляющей AMN-семантику для

этой модели. Абстрактная машина Mch_M для модели данных M должна быть построена таким образом, чтобы множество схем модели M находилось в биекции с множеством допустимых состояний машины Mch_M, ограниченным ее инвариантом I_M. Отображение из исходной модели в целевую сводится к моделированию понятий целевой модели посредством понятий исходной модели. Для этой цели строится AMN-машина, являющаяся расширением AMN-машины исходной модели и уточнением AMN-машины целевой модели данных.

Принципы формализации моделей данных

Предлагаются следующие принципы AMN-формализации моделей данных:

1. Базовые понятия моделей данных формализуются с помощью системы типов AMN;
2. Сложные понятия моделей данных формализуются с помощью абстрактных машин.

Для определения AMN-семантики моделей данных сперва формализуются базовые понятия этих моделей. Рассмотрим формализацию основных понятий реляционной и XDM моделей данных посредством системы типов AMN. Ниже приведена формализация основных понятий реляционной модели, а именно схемы отношения и отношения. Схема отношения характеризуется конечным множеством атрибутов и ограничений, определенных над этой схемой. Примерами ограничений являются: ключ, внешний ключ, ограничения над атрибутами, и т.д. В данном контексте предлагается следующая формализация понятия схемы отношения посредством системы типов AMN:

```
RelationSchemaType = struct(
        attrset : ℙ(String × AtomicType),
        key : ℙ(String),
        foreignkey : struct(
                attrset : ℙ(String),
                referencing : struct(
                        name : String,
                        attrset : ℙ(String))),
```

constraint : Boolean);

Здесь и далее, множество базовых типов (например, целочисленный, строковой, и т.д.) представлено множеством *AtomicType*. Пусть R – схема отношения. Отношение, соответствующее схеме R, представляет собой конечное множество кортежей. Отношение характеризуется именем, схемой и отображением. Далее следует формализация понятия отношения посредством системы типов AMN:

RelationType = struct(
name : String,
schema : RelationSchemaType,
relation : $\mathbb{P}(\times_{s \in RelationSchemaType.attrset}$ *ran*(*s*))
);

Теперь рассмотрим формализацию основных понятий модели данных XDM, а именно – схемы и элемента. Схема характеризуется пятеркой *name* (имя элемента), *image* (наименование базы данных исходной модели), *typingop* (конструктор элемента), *content* (определение содержимого элемента) и *frequency* (частота вхождения элемента). Предлагается следующее рекурсивное определение понятия схемы:

SchemaType = struct(
name : string,
image : string,
typingop : {*sequence, choice, all, none*},
content : AtomicType × *Frequency* | *seq*(*String* × *SchemaType* × *Frequency*)
frequency : Frequency
);

Здесь и далее Frequency = $\{?, *, +, \perp\}$. Элемент характеризуется своей схемой и отображением, согласно определениям 2.1.1 и 2.1.2. Понятие элемента формализуется посредством системы типов AMN следующим образом:

ElementType = struct(
schema : SchemaType,
element : $\mathbb{P}$(*SchemaType*)
);

AMN-машина для реляционной модели данных

Состояние этой абстрактной машины определяется посредством двух переменных: $rdbs$ (схема реляционной базы данных) и rdb (реляционная база данных). Инвариант определяет общие свойства машины. В данном случае он утверждает что $rdbs$ является подмножеством множества схем реляционной базы данных, а rdb является подмножеством множества отношений реляционной базы данных т.ч. для каждой схемы $s \in rdbs$ существует $r \in rdb$ т.ч r является отношением над s и удовлетворяет заданным ограничениям. Рассмотрены следующие виды ограничений: ограничения по ключу, ограничения по внешнему ключу и ограничения над кортежами. В секции операций представлена формализация операций реляционной алгебры, посредством которых можно изменять состояние машины. Так как реляционная база данных является экземпляром данной машины, то с помощью этих операций можно изменять состояния реляционной базы данных. Ниже представлена абстрактная машина для реляционной модели:

MACHINE

$RelationalDataModel$

VARIABLES

$rdbs, rdb$

INVARIANT

$$rdbs \in \mathbb{P}(RelationSchemaType) \wedge rdb \in \mathbb{P}(RelationType) \wedge$$
$$(\forall s \in rdbs \, \exists r \in rdb \cdot (r.schema.attrset = s) \wedge$$
$$\forall a \in r.schema.attrset \cdot (dom(a) \in ran(a)) \wedge$$
$$r.schema.key \subseteq dom(r.schema.attrset) \wedge$$
$$constraint \in \mathbb{P}\big(dom(r.schema.attrset)\big) \rightarrow Boolean \wedge$$
$$(\forall r_1 \in rdb \, \exists r_2 \in rdb \cdot r_1.schema.foreignkey.attrset \neq \emptyset \wedge r_1.schema.foreignkey.referencing.name = r_2.name \wedge r_2.schema.key.attrset = r_1.schema.foreignkey.attrset))$$

INITIALISATION

$$rdbs, rdb := \{\ldots\}, \{\ldots\}$$

OPERATIONS

$$res \leftarrow RelationalUnion(r_1, r_2) =$$

PRE

$r_1 \in rdb \wedge r_2 \in rdb \wedge unionCompatible(r_1.schema, r_2.schema)$

THEN

$res := rec(schema : r_1.schema, tuples : r_1.tuples \cup r_2.tuples)$

END

…

END

В секции операций представлено определение операции объединения реляционной алгебры. Предикат $unionCompatible$ гарантирует, что объединяемые отношения обладают совместимыми для объединения схемами.

AMN-машина для XDM модели данных

Как и в случае с абстрактной машиной для реляционной модели данных, состояние этой машины представлено посредством двух переменных: $xdbs$ (схема XDM базы данных) и xdb (XDM база данных). Секция инварианта определена аналогично случаю реляционной модели. В данном случае инвариант утверждает, что $xdbs$ является подмножеством множества схем XDM баз данных, а xdb является подмножеством множества XDM–элементов таким образом, что для любого $s \in xdbs$ существует $e \in xdb$ т.ч. e является элементом над s. В секции операций представлена формализация операций алгебры элементов. Посредством этих операций можно изменять состояние базы данных. Предложена следующая формализация XDM модели данных:

MACHINE

XDM

VARIABLES

$xdbs, xdb$

INVARIANT

$xdbs \in \mathbb{P}(SchemaType) \wedge xdb \in \mathbb{P}(ElementType) \wedge$
$(\forall s \in xdbs \; \exists e \in xdb \cdot identically(e.schema, s.schema)$

INITIALISATION

$xdbs, xdb := \{\ldots\}, \{\ldots\}$

OPERATIONS

$res \leftarrow XDMUnion(e_1, e_2) =$

PRE

$e_1 \in xdb \wedge e_2 \in xdb \wedge similar(e_1.schema, e_2.schema)$

THEN

$res := rec(schema : e_1.schema \oplus e_2.schema,$
$element : e_1.element \cup e_2.element)$

END

...

END

Рассмотренный предикат $identically$ в секции инварианта имеет место, если схемы элементов e и s идентичны. Предикат $similar$ в предусловии операции является предикатом, гарантирующим что операция объединения будет выполнена только для подобных схем.

Расширение канонической модели

Пусть построены абстрактные машины Mch_T для канонической модели, Mch_S для модели данных источника. Каноническая модель расширяется с помощью машины Ext_{ST}, содержащей понятия исходной модели, отсутствующие в целевой. Эти понятия представлены в виде множеств и констант машины Ext_{ST}. Ниже приведены AMN-представления исходной и канонической моделей:

MACHINE Mch_T
SETS $Sets_T$
CONSTANTS $Const_T$
PROPERTIES P_T
VARIABLES Var_T
INITIALISATION $Init_T$
INVARIANT I_T
OPERATIONS Op_T
END

MACHINE Mch_S
SETS $Sets_S$
CONSTANTS $Const_S$
PROPERTIES P_S
VARIABLES Var_S
INITIALISATION $Init_S$
INVARIANT I_S
OPERATIONS Op_S
END

Следующая AMN-машина представляет собой расширение канонической модели:

MACHINE Ext_{ST}
EXTENDS Mch_T
SETS $Sets_{ST}$
CONSTANTS $Const_T$

PROPERTIES P_{ST}
END

Уточнение канонической модели

Пусть для исходной модели M_S и канонической модели M_T уже построены соответствующие абстрактные машины. В рамках предлагаемого подхода к интеграции данных, для доказательства корректности отображения из исходной модели в каноническую, строится абстрактная машина, которая расширяет AMN-машину исходной модели и уточняет AMN-машину канонической модели. В секции инварианта данной машины определяются условия соответствии схем исходной и канонической моделей. В секции операций посредством операций модели источника моделируются операции алгебры элементов. Ниже представлена AMN-схема для этой машины:

REFINEMENT $Ext_{ST}Ref$
REFINES Ext_{ST}
EXTENDS Mch_S
INVARIANT I_R
OPERATIONS Op_R
END

Если теперь для этого уточнения выполняются условия, приведенные в определении 1.3.1, то можно заключить, что AMN-семантика исходной модели представляет собой уточнение AMN-семантики исходной модели. Таким образом доказывается корректность построенного отображения.

Реляционное уточнение канонической модели

В этой секции рассматривается реляционное уточнение ядра канонической модели. Без нарушения общности предполагается, что в рамках канонической модели для построения сложных типов используется конструктор *sequence*. Помимо переменных машины *RelationalDataModel* используется три дополнительные переменные, определяющие состояние этой машины. В секции инварианта определены семантика этих переменных и условия соответствия реляционной и

канонической схем. AMN-машина для реляционного уточнения канонической модели строится следующим образом:

REFINEMENT $XDMRef$
REFINES XDM
EXTENDS $RelationalDataModel$
VARIABLES
$rref1, rref2, resref$
INVARIANT
$rref1 \in rdb \wedge rref2 \in rdb \wedge resref \in rdb \wedge \forall s \in xdbs \cdot ($
$s.image =' rdbs' \wedge \exists r \in rdb \cdot (s.name = r.name \wedge$
$(s.typing = none \wedge card(r.schema.attrset) = 1 \wedge$
$dom(s.content) = dom(r.schema.attrset) \wedge$
$ran(s.content = \perp \vee s.typing = sequence \wedge$
$card(r.schema.attrset) = size(s.content) \wedge s.schema.frequency = \perp \wedge$
$\forall i \in [1, size(s.content)] \wedge \exists a \in r.schema.attrset \cdot ($
$dom(a) = dom\left(dom\left(s.content(i)\right)\right) \wedge$
$ran(a) = ran(dom(s.content))))) \wedge s.frequency = \perp)$
OPERATIONS
$res \leftarrow XDMUnion(e_1, e_2) =$
PRE
$e_1 \in xdb \wedge e_2 \in xdb \wedge similar(e_1.schema, e_2.schema)$
THEN
$rref1 := e_1 \,||\, rref2 := e_2$
$resref := RelationalUnion(rref1, rref2)$
$res := resref$
END
...
END

В секции *OPERATIONS* операция $union$ алгебры элементов смоделирована посредством аналогичной операции реляционной алгебры и операторов присваивания. Посредством AMN-программ были аналогичным образом представлены остальные операции реляционной алгебры, определены правила преобразования элемента в отношение и обратно. С использованием программного обеспечения Atelier-B [82] была доказана корректность построенного отображения.

Идея построения канонической схемы

В предлагаемом подходе к интеграции информации формализм AMN имеет двойное применение. Во-первых, он используется для доказательства корректности отображения из исходной модели в каноническую. Во-вторых, на основе AMN-машин исходной и канонической моделей и машины-уточнения канонической модели можно генерировать канонические схемы.

Для поддержки канонических схем предполагается разработка словаря данных (в виде XML-приложения) для канонической модели. Словарь данных должен содержать три типа метаданных:

1. Метаданные в контексте традиционных баз данных (данные об объектах канонической модели);
2. Метаданные для присвоения неформальной и формальной семантики всем понятиям, используемым в канонической модели;
3. Метаданные для формализации сигнатур понятий канонической модели (для проверки семантической корректности представления объектов канонической модели).

Таким образом, словарь данных канонической модели = DTD для канонической модели + DTD для понятий канонической модели + DTD для сигнатур понятий канонической модели. В данном контексте целью является создание эффективных алгоритмов для генерации экземпляров DTD канонической модели.

2.5 Выводы по главе

В главе 2, в рамках предлагаемого подхода к интеграции данных, разработана каноническая модель данных, основанная на XML.

- Обоснован выбор модели данных XDM в качестве ядра канонической модели;
- Предложен подход к интеграции данных, основанный на онтологии;
- Предложено формальное описание понятий интеграции данных посредством XML;
- Предложен принцип расширения ядра канонической модели;
- Предложены правила AMN-формализации моделей данных;
- Построены AMN-машины для канонической и реляционной моделей данных и их обратимого отображения и доказана его корректность;
- Предложена идея построения канонической схемы.

3 ХРАНИЛИЩЕ ДАННЫХ ДЛЯ КАНОНИЧЕСКОЙ МОДЕЛИ ДАННЫХ

Материализованная интеграция данных предполагает построение хранилища данных. В данной главе предлагается подход к построению хранилища данных, в основном ориентированный на поддержку OLAP приложений. Важность использования технологий хранилищ данных в приложениях OLAP обусловлена несколькими причинами [10]. Прежде всего, хранилище данных является необходимым средством организации и централизации корпоративной информации с целью поддержки запросов OLAP (исходные данные нередко распределены по многим неоднородным источникам). Но чаще еще более значимым оказывается тот факт, что запросы OLAP, весьма сложные по природе и затрагивающие крупные массивы данных, требуют чрезмерно больших временных затрат на выполнение в среде традиционных систем обработки транзакций. Поэтому такие процессы принято относить к категории «длинных» транзакций. Длинные транзакции, «блокирующие» базу данных на длительное время, существенно снижают вероятность успешного выполнения ординарных операций OLTP (on-line transaction processing). Одно из наиболее практичных решений состоит в копировании первичной информации в хранилище данных и выполнении запросов OLAP исключительно в среде хранилища, а запросов и операций модификации, попадающих в категорию OLTP,- в системах, обслуживающих источники данных. Обычной практикой является обновление содержимого хранилища данных ночью – на следующий день аналитики получают возможность пользоваться наиболее «свежей» статической копией первичной информации. Таким образом, данные в хранилище «устаревают», как правило, не более чем на 24 часа, но степень их актуальности в контексте большинства приложений поддержки принятия решений остается вполне приемлемой.

Концепция сеточных фалов [44], [45] является одним из адекватных формализмов эффективного управления многомерными данными [10] и может быть использована для эффективного хранения кубов данных в хранилищах [83], [84]. Модель сеточного файла може представлять так, будто пространство точек-объектов

разбивается на части воображаемой сеткой. Линии сетки, параллельные осям координат, разделяют пространство на *полосы* (stripes). Количество линий сетки по различным измерениям может варьироваться, также может различаться ширина полос – даже в пределах одного измерения. Пересечения этих полос образуют прямоугольные области, именуемые *ячейками* сеточного файла. Принадлежащие ячейке точки представляются записями, хранящимися в соответствующем ей *блоке*, указатель на который в свою очередь хранится в ячейке.

На рис. 3.1 приведен пример 3-мерного сеточного файла. Здесь X, Y и Z являются измерениями рассматриваемого пространства, которое разделено на полосы v_1, v_2, v_3 в измерении X, w_1, w_2 в измерении Y и u_1, u_2, u_3 в измерении Z.

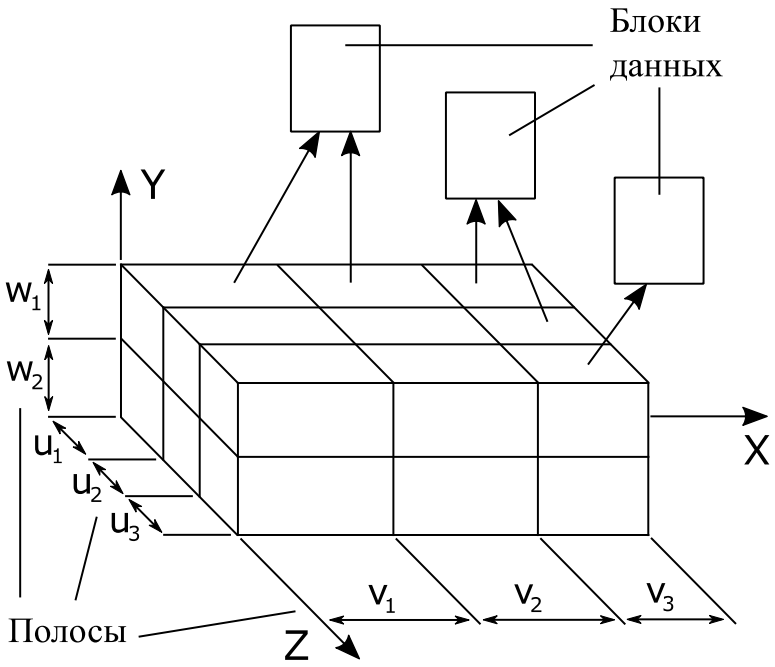


Рис. 3.1. Пример 3-мерного сеточного файла

Динамические аспекты файловых структур, где все ключи рассматриваются как симметричные, избегая различий между первичными и вторичными ключами, изучены в [44]. Статья представляет понятия сеточного разделения пространства поиска и директории сетки, являющиеся ключевыми в динамической структуре, именуемой сеточным файлом. Эта структура способна адаптироваться к своему содержимому при операциях вставки и удаления, достигая таким образом верхней границы в две

дисковые операции для поиска одного значения. Она также эффективно обрабатывает запросы на интервалах и частично специфицированные запросы. Рассмотрены некоторые подходы к разделению и объединению ячеек, приводящие к различным детализациям сеточного разделения.

В [85] приведены алгоритмы, обобщающие стандартные методы поиска по ключу и применяющие их к поиску записей по нескольким ключам. Рассмотрены также два подхода к организации индексных файлов, многомерное динамическое хеширование и многомерное расширяемое хеширование, являющиеся многомерными обобщениями динамического и расширяемого хеширования соответственно, а также оценены средние размеры индексов. Многомерные расширения линейных и расширяемых хеш-таблиц также приводятся в [10], [86], [87].

В данной работе предлагается расширение концепции сеточного файла. Каждая полоса рассматривается как линейная хеш-таблица, что позволяет плавнее наращивать количество сегментов. Также предлагается структура внутреннего представления сеточного файла в виде ориентированного ациклического графа, имеющая целью уменьшить объем директории индекса. Разработан язык определения данных, независимый от парадигм управления данными. Для решения проблемы пустых ячеек в сеточном файле используется механизм *сегментирования* (chunking). Описаны операции над сеточными файлами и приведены оценки их сложности в контексте модификации индекса и дисковых операций.

3.1 Модификация структуры сеточного файла

Одним из недостатков, присущих сеточным файлам, является проблема неэффективного использования памяти группами ячеек, ссылающихся на одни и те же блоки данных. В данной работе предлагается альтернативная структура индекса, основанная на концепции сеточного файла и имеющая целью избежать хранения повторяющихся указателей на одни и те же блоки данных, а также поддерживать

плавный рост размера директории индекса и обеспечить разумные стоимости операций.

В данном подходе сеточный файл не хранится в виде многомерного массива. Причина этого заключается в том, что при каждом разделении сегмента данных одна из пересекающих его полос также разделяется на две полосы, таким образом удваивая количество ячеек исходной полосы, при этом многие из новых ячеек содержат повторяющиеся указатели на одни и те же блоки данных. Вместо этого, все ячейки, для которых соответствующие записи хранятся в одном и том же блоке данных, объединяются в *сегменты* (chunks), представленные едиными ячейками памяти с одним указателем на соответствующий блок. Сегменты являются основными единицами ввода/вывода данных, а также используются для кластеризации данных. Сегменты используются в качестве механизма разрешения проблемы пустых ячеек в сеточном файле. Для каждого измерения информация о его разделении хранится в линейной шкале, каждый элемент которой соответствует полосе сеточного файла и представляется в виде массива указателей на пересекаемые этой полосой сегменты.

Каждая полоса рассматривается в качестве линейной хеш-таблицы. С целью уменьшения количества сегментов сеточного файла используются блоки переполнения. Количество блоков переполнения может различаться среди сегментов. При этом оно поддерживается на таком уровне, чтобы для каждой полосы среднее количество блоков переполнения для пересекаемых ею сегментов было меньше единицы [88]. Это позволяет значительно уменьшить общее количество сегментов, гарантируя выполнение не более двух дисковых операций для доступа к данным.

На рис. 3.2 приведен пример сеточного файла, построенного с применением предложенной модифицированной структуры.

Используются стратегии слияния и разделения, схожие с описанными в [44]. Измерения для обеих операций выбираются таким образом, чтобы количество полос в различных измерениях различалось не более чем на единицу. При разделении сегмента координата делящей гиперплоскости выбирается таким образом, чтобы количества точек, попадающих в полученные в результате разделения сегменты,

различались как можно меньше. Для этой цели для каждого сегмента хранятся статистические данные о средних значениях координат находящихся в нем точек.

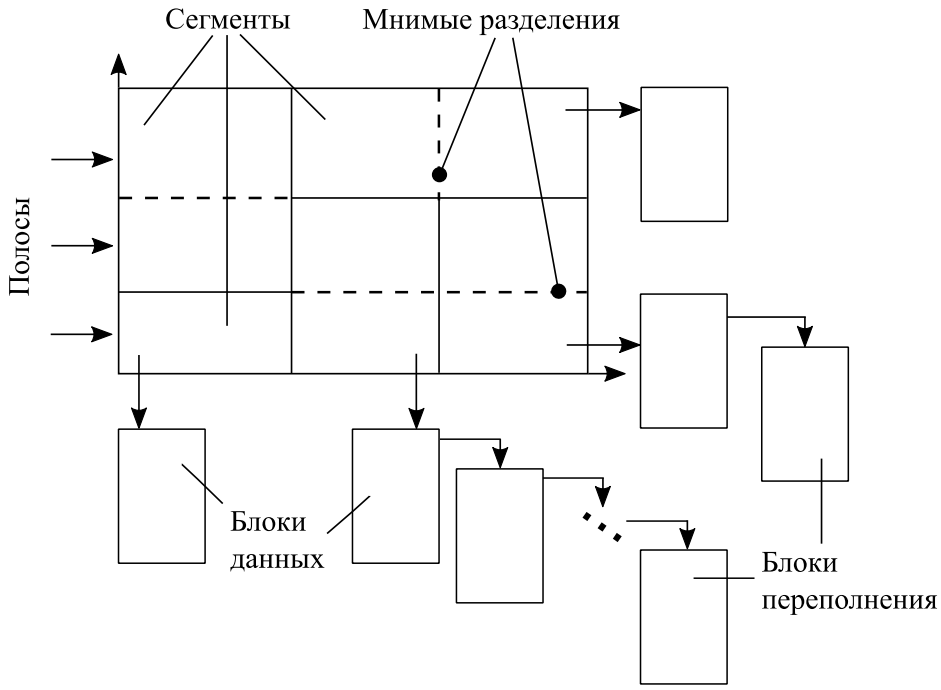


Рис. 3.2. Пример 2-мерного модифицированного сеточного файла

Слияние осуществляется в тех случаях, когда заполненность результирующего сегмента примерно равно 70 процентам. Это значение получено эмпирическим путем, и применяется с целью достижения приемлемой производительности [44].

3.1.1 Формализация предложенной концепции

Сеточный файл формально представляется в виде тройки $F = \langle D, S, C \rangle$ где D есть множество измерений, S – множество полос, а C – множество сегментов. Каждая полоса соответствует в точности одному измерению и пересекает некоторое непустое подмножество множества сегментов.

В данном подпараграфе рассматриваются некоторые характеристики модифицированной структуры сеточных файлов, и приводятся оценки их величин.

Анализ проводится при предположении, что все измерения поля данных сеточного файла являются независимыми и эквивалентными (равноправными) [44], [85]. Количество измерений обозначается как *n*, а среднее количество операций разделения, осуществленных в каждом измерении, как *m*.

Количество ячеек в сетке. Так как каждое из *n* измерений разделено на *m* частей в среднем, общее количество ячеек в структуре сеточного файла в среднем равно m^n.

Количество полос в одном измерении. При проведении одной операции разделения может возникнуть не более одной новой полосы. Количество полос может быть уменьшено при операциях слияния. Таким образом, количество полос в одном измерении ограничено сверху количеством операций разделения, осуществленных в данном измерении, и имеет порядок $O(m)$.

Общее количество полос таким образом имеет порядок $O(nm)$.

Общее количество сегментов. Новые сегменты возникают только при осуществлении операций разделения, при этом в результате одной операции разделения количество сегментов увеличивается ровно на единицу. Это означает, что общее количество сегментов ограничено сверху количеством произведенных операций разделения и может быть оценено как $O(nm)$.

Среднее количество ячеек в сегменте является отношением общего количества ячеек к количеству сегментов и имеет порядок

$$O\left(\frac{m^n}{nm}\right) = O\left(\frac{m^{n-1}}{n}\right)$$

Средняя величина стороны сегмента. Для оценки в качестве единицы измерения используется средняя величина стороны ячейки сетки. Без нарушения общности можно предположить, что средней формой сегмента является n-мерный куб. В таком случае средняя величина его стороны будет иметь порядок

$$O\left(\sqrt[n]{\frac{m^{n-1}}{n}}\right) = O\left(\frac{m}{\sqrt[n]{nm}}\right)$$

Среднее количество сегментов, пересекаемых полосой. Полоса имеет среднюю длину в m ячеек в $n-1$ измерении. Так как средняя величина стороны сегмента имеет порядок

$$O\left(\frac{m}{\sqrt[n]{nm}}\right)$$

то количество сегментов, пересекаемых полосой, будет иметь порядок

$$O\left(\left(\frac{m}{\frac{m}{\sqrt[n]{nm}}}\right)^{n-1}\right) = O\left(\left(\sqrt[n]{nm}\right)^{n-1}\right)$$

Для простоты дальнейших рассуждений мы ослабим эту оценку до $O(nm)$.

Размер директории индекса. Так как каждая из $O(nm)$ полос пересекает в среднем $O(nm)$ сегментов, общее количество хранимых указателей будет иметь порядок $O(n^2m^2)$. Также, каждый сегмент хранит один указатель на соответствующий блок данных. Общее количество таких указателей - $O(nm)$. Таким образом, величина директории индекса имеет порядок $\boldsymbol{O(n^2m^2)}$.

3.1.2 Операции над сеточными файлами

В данном подпараграфе рассматриваются операции над сеточными файлами, адаптированные к применению над предложенной модифицированной структурой. Приводятся сложности операций в контексте модификации индекса и дисковых операций.

Поиск по ключу. Для нахождения всех записей по значению d_0 измерения d сперва необходимо найти полосу s этого измерения, которой принадлежит d_0, затем рассмотреть все сегменты, пересекаемые полосой s. Для каждого из них необходимо выполнить чтение соответствующих блоков данных и проверить хранящиеся в них записи на соответствие запросу. Так как все эти сегменты должны быть рассмотрены, и никакие другие сегменты не могут содержать результатов запроса, только необходимые и достаточные сегменты будут просмотрены. Согласно предыдущему подпараграфу, среднее количество таких сегментов имеет порядок $O(nm)$.

Соответствующая заданной координате полоса может быть найдена за время $O(\log m)$ с помощью бинарного поиска по упорядоченной линейной шкале данного измерения. Сложность одной операции поиска, таким образом, будет иметь порядок $\boldsymbol{O(nm \log m)}$. Помимо этого, возможно использование эффективных структур данных, таких как деревья ван Эмде Боаса [89], [90]. В §4.1 описывается метод использования деревьев ван Эмде Боаса в случае, когда координаты точек представляются в виде 32-битных целых чисел, что позволяет существенно сократить количество операций при поиске полосы по сравнению с логарифмическими алгоритмами. Количество дисковых операций, в силу использования блоков переполнения, в среднем равно удвоенному количеству рассматриваемых сегментов и имеет порядок $\boldsymbol{O(nm)}$. В §4.1 также описан метод применения хеш-функций для исключения из рассмотрения сегментов, заведомо не содержащих удовлетворяющих запросу записей, что позволяет сократить количество дисковых операций.

Ниже приведен псевдокод операции поиска по ключу:

Алгоритм 3.1.1. Поиск значений по ключу

Find(v d):
in: v – ключ поиска, d – измерение
out: множество записей, соответствующих критерию поиска
1: $result \leftarrow \emptyset$
2: $s \leftarrow$ FindStripe(d, v)
3: **for all** $c \in$ ChunksOfStripe(s) **do**
4: $b \leftarrow$ LoadFirstBlock(c)
5: $blockExists \leftarrow TRUE$
6: **while** $blockExists$ **do**
7: **for all** $r \in b$ **do**
8: **if** $\pi_d(r) = v$ **then**
9: $result \leftarrow result \cup \{r\}$
10: **end if**
11: **end for**
12: **if** ExistsNextBlock(b) **do**
13: $b \leftarrow$ LoadNextBlock(b)
14: **else**
15: $blockExists \leftarrow FALSE$

16: **end if**
17: **end while**
18: **end for**
19: **return** $result$

Поиск с совпадением отдельных координат. При поиске по нескольким координатам для сперва находятся полосы, соответствующие заданным координатам, далее производится пересечение множеств пересекаемых ими сегментов и рассматриваются только блоки данных, соответствующие сегментам пересечения. Аналогично выводам, приведенным в описании операции поиска по одному ключу, получаем, что нахождение полос занимает $O(k \log m)$ времени. Для пересечения множеств указателей на сегменты рассматривается множество сегментов, пересекаемых одной полосой, и для каждого из них проверяется, имеет ли место пересечение с остальными рассматриваемыми полосами. Это занимает $O(k)$ времени, так как проверка на пересечение сегмента и полосы осуществляется за константное время путем сравнения координат их границ. Таким образом, сложность операция пересечения имеет порядок $O(knm)$, и общая сложность операции поиска по k координатам также имеет порядок $O(k \log m + knm)$=$\boldsymbol{O(knm)}$.

Алгоритм 3.1.2. Поиск с совпадением отдельных координат

Find(V, D):
in: V – последовательность ключей поиска $\langle v_0, \dots, v_{k-1} \rangle$
D – последовательность измерений $\langle d_0, \dots, d_{k-1} \rangle$
out: множество записей, соответствующих критерию поиска
1: $result \leftarrow \emptyset$
2: $s \leftarrow$ FindStripe(d_0, v_0)
3: $chunks \leftarrow$ ChunksOfStripe(s)
4: **for all** $c \in chunks$ **do**
5: **for** $i \leftarrow 1 \dots (k-1)$ **do**
6: $s \leftarrow$ FindStripe(d_i, v_i)
7: **if** LowerBound(s) > UpperBound(c, d_i) **or**
UpperBound(s) < LowerBound(c, d_i) **then**
8: **break**

```
            end if
        if i < k do
            chunks ← chunks \ {c}
        end if
    end for
    for all c ∈ chunks do
        b ← LoadFirstBlock(c)
        blockExists ← TRUE
        while blockExists do
            for all r ∈ b do
                if π_d(r) = v then
                    result ← result ∪ {r}
                end if
            end for
            if ExistsNextBlock(b) do
                b ← LoadNextBlock(b)
            else
                blockExists ← FALSE
            end if
        end while
    end for
    return result
```

Для осуществления пересечения множеств указателей на пересекаемые полосами сегменты возможно использование алгоритмов быстрого пересечения множеств в памяти [91], позволяющих достичь эффективной оценки

$$O\left(\frac{\sum_{i=1}^{n}|s_i|}{\sqrt{w}}\right)$$

операций для пересечения множеств s_i, где w есть размер машинного слова. Предпосылкой для возможности использования подобных алгоритмов является тот факт, что множества указателей на пересекаемые полосами сегменты обновляются только во время операций разделения и слияния сегментов, что происходит значительно реже регулярных операций поиска, вставки и удаления, и позволяет хранить эти множества в виде удобных для этих алгоритмов структур. В данном случае, размер каждого из k рассматриваемых множеств указателей, согласно §3.1.1, в среднем имеет порядок $O(nm)$. Таким образом, использование алгоритмов быстрого

пересечения множеств позволяет сократить сложность операции поиска по k ключам до

$$\boldsymbol{O\left(\frac{knm}{\sqrt{w}}\right)}.$$

Стоит отметить, что дальнейшая модификация структуры сеточного файла, приведенная в §3.2, делает затруднительным применение подобных алгоритмов.

Поиск заданной точки представляет собой частный случай поиска по нескольким ключам, при котором в запросе присутствуют все n измерений. Имея $k = n$, получаем, что сложность операции поиска заданной точки составляет $\boldsymbol{O(n^2m)}$, а использование алгоритмов быстрого пересечения множеств для нахождения искомого сегмента позволяет сократить эту оценку до

$$\boldsymbol{O\left(\frac{n^2m}{\sqrt{w}}\right)},$$

где w – размер машинного слова. Так как в данном случае будет рассмотрен ровно один сегмент, содержащий заданную точку, количество дисковых операций в среднем не будет превосходить 2.

Поиск ближайших соседних объектов. Для нахождения точек, ближайших к заданной, необходимо найти сегмент, которому данная точка принадлежит, и сравнить записи, находящиеся в соответствующих ему блоках данных. Однако возможны ситуации, когда ближайшая точка принадлежит одному из соседних сегментов (например, когда заданная точка находится очень близко к границе сегмента), и они также подлежат рассмотрению. В худшем случае будут рассмотрены $2n$ соседних сегментов. Таким образом, сложность операции поиска ближайших соседних объектов не будет сильно отличаться от сложности операции поиска заданной точки. Однако количество дисковых операций при рассмотрении соседних сегментов может сильно возрасти – в худшем случае потребуется $2(2n + 1)$ дисковых операций.

Поиск в диапазонах значений. Алгоритм аналогичен алгоритму поиска по нескольким координатам, но в данном случае в каждом измерении рассмотрению может подлежать более чем одна полоса. Это делает проблематичным применение алгоритмов быстрого пересечения множеств, так как множества сегментов,

пересекаемых полосами одного измерения, предварительно необходимо объединить. Предполагая, что рассматриваемые диапазоны значений будут охватывать в среднем t полос в каждом измерении, аналогично рассуждениям, приведенным при оценке операции поиска по k ключам, получим, что сложность операции поиска в диапазонах значений будет равна $\boldsymbol{O(tnmk)}$. Количество дисковых операций в среднем будет равно удвоенному размеру полученного пересечения множеств сегментов.

Алгоритм 3.1.3. Поиск с совпадением отдельных координат

Find(L, R, D):

in: L – последовательность левых границ диапазонов $\langle l_0, \dots, l_{k-1} \rangle$
R – последовательность правых границ диапазонов $\langle r_0, \dots, r_{k-1} \rangle$
D – последовательность измерений $\langle d_0, \dots, d_{k-1} \rangle$

out: множество записей, соответствующих критерию поиска

1: $result \leftarrow \emptyset$
2: $chunks \leftarrow \emptyset$
3: $stripes \leftarrow$ FindStripes(d_0, l_0, r_0)
4: **for all** $s \in stripes$ **do**
5: $chunks \leftarrow chunks \cup$ ChunksOfStripe(s)
6: **end for**
7: **for all** $c \in chunks$ **do**
8: **for** $i \leftarrow 1 \dots (k-1)$ **do**
9: **if** $l_i >$ UppwerBound(c, d_i) **or** $r <$ LowerBound(c, d_i) **then**
10: **break**
11: **end if**
12: **if** i < k **do**
13: chunks $\leftarrow chunks \setminus \{c\}$
14: **end if**
15: **end for**
16: **for all** $c \in chunks$ **do**
17: $b \leftarrow$ LoadFirstBlock(c)
18: $blockExists \leftarrow TRUE$
19: **while** $blockExists$ **do**
20: **for all** $r \in b$ **do**
21: **if** $\pi_d(r) = v$ **then**
22: $result \leftarrow result \cup \{r\}$
23: **end if**

```
24:             end for
25:             if ExistsNextBlock(b) do
26:                 b ← LoadNextBlock(b)
27:             else
28:                 blockExists ← FALSE
29:             end if
30:         end while
31:     end for
32:     return result
```

Операция вставки. При вставке записи сначала, согласно координатам соответствующей ей точки, необходимо определить сегмент, которому она принадлежит, затем загрузить соответствующий блок данных и осуществить вставку. Нахождение сегмента осуществляется аналогично операции поиска заданной точки. Так как каждая полоса рассматривается как линейная хеш-таблица, в случае недостатка свободной памяти в блоке возможно добавление блока переполнения. Однако, если после вставки возникает полоса, для которой среднее количество блоков переполнения в каждом пересекаемом ею сегменте превосходит 1, необходимо выполнить разбиение сегмента на две части и реорганизовать соответствующие блоки данных.

Следующий псевдокод иллюстрирует процедуру вставки.

Алгоритм 3.1.4. Вставка значения

```
Insert(v):
    in: v – значение, подлежащее вставке
    1:  c ← FindChunk(v)
    2:  b ← LoadLastBlock(c)
    3:  if not Full(b) or CanOverflow(c) then
    4:      AddRecordToBlock(v, b)
    5:  else
    6:      Split(c)
    7:      Insert(v)
    8:  end if
```

Функция AddRecordToBlock осуществляет непосредственную вставку данного значения в блок. Если это невозможно ввиду недостатка памяти, создается блок переполнения и значение добавляется в него.

Алгоритм 3.1.5. Добавление записи в блок

AddRecordToBlock(v, b):
in: v - значение, подлежащее вставке,
b – блок, в который нужно добавить значение
out: блок, в который было в действительности добавлено значение
1: **if** Full(b) **then**
2: AddOverflowBlock(b)
3: $b \leftarrow$ LoadNextBlock(b)
4: **end if**
5: WriteRecordToBlock(v, b)
6: SaveBlock(b)
7: **return** b

Операция разделения. Пусть необходимо провести разделение сегмента c. Во-первых, необходимо выбрать измерение, в котором будет осуществлено разделение. Для этой цели выбирается такое измерение d, которое на данный момент состоит из наименьшего количества полос. Это делается с целью поддержки симметричности структуры сеточного файла относительно измерений. Далее выбирается некоторая координата d_0 в измерении d, которая определяет гиперплоскость деления. Затем сегмент c разделяется на два сегмента c_1 и c_2, при этом все записи из c, для которых соответствующие точки имеют координату в измерении d меньшую, чем d_0, сохраняются в сегменте c_1, а все остальные – в сегменте c_2. Координата d_0 выбирается таким образом, чтобы разница между количеством записей в результирующих сегментах была как можно меньше. В результате разделения сегмента некоторая полоса s измерения d также разделяется на полосы s_1 and s_2. Указатели на новые сегменты должны быть надлежащим образом добавлены к спискам указателей пересекающих их полос. Согласно §3.1.1 списки указателей новых полос будут состоять из $O(nm)$ элементов. Также $O(nm)$ указателей на новые сегменты будут добавлены в списки указателей полос других измерений. Таким образом, операция разделения будет иметь

порядок $\boldsymbol{O(nm)}$. Для распределения записей в среднем потребуется не более 2 дисковых операций для чтения данных сегмента c, и не более 2 дисковых операций для каждого из сегментов c_1 и c_2. Таким образом, среднее количество дисковых операций при осуществлении разделения сегмента не превосходит 6.

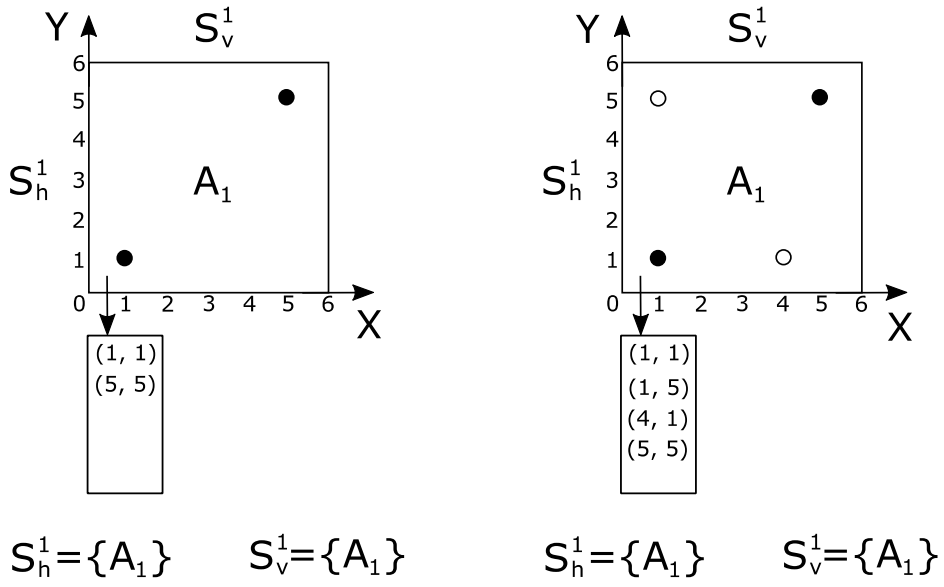
Псевдокод операции разделения представлен Алгоритмом 3.1.6:

Алгоритм 3.1.6. Разделение сегмента

```
Split(c):
    in: c – сегмент, подлежащий разделению
    Найти измерение для разделения d координату разделения d_0
    Проинициализировать новые сегменты c_1 и c_2
    Загрузить первые блоки c, c_1 и c_2 как b, b_1 и b_2 соответственно
    repeat
        for all r ∈ b do
            if π_d(r) ≤ d_0 then
                b_1 ← AddRecordToBlock(r, b_1)
            else
                b_2 ← AddRecordToBlock(r, b_2)
            end if
        end for
        b ← LoadNextBlock(b)
    until Exists(b)
    s ← FindStripe(d, d_0)
    Разделить полосу s по координате d_0 на две полосы s_1 и s_2
    for all c_0 ∈ StripeChunks(s) do
        Добавить указатель на сегмент c_0 в полосы s_1 и s_2
    end for
    Добавить указатели на сегменты c_1, c_2 в полосы s_1, s_2 соответственно
    for all s ∈ StripesOfChunk(c) do
        Добавить указатели на сегменты c_1 и c_2 в полосу s
    end for
    RemoveChunk(c)
```

В качестве примера рассмотрим 2-мерный сеточный файл с координатными осями X и Y. Пусть каждый блок имеет объем памяти, достаточный для хранения не более чем 5 записей. По аналогии с линейными хеш-таблицами, будем следить за тем, чтоб в каждой полосе отношение среднего числа записей в блоке к размеру блока не превосходило некоторого константного значения [10]. В качестве такого значения будем использовать величину 80%. Обозначим количество записей в полосе через r а количесво сегментов, пересекаемых полосой, через c. Исходя из выбранной выше величины 80% получим, что для каждой полосы должно иметь место неравенство $r/c \leq 4$. Обозначим отношение r/c через k.

Изначально имеется единственный сегмент A_1, содержащий две записи (1; 1) и (5; 5). Также имеется одна горизонтальная полоса S_h^1 и одна вертикальная полоса S_v^1. Далее добавляются две точки (1; 5) и (4; 1). После этого для каждой полосы имеет место $r = 4$; $c = 1$ и $k = 4$. Таким образом процедура может быть продолжена без разделения сегмента. Так как в блоке данных, соответствующем рассматриваемому сегменту, достаточно свободной памяти для хранения всех добавляемых значений, необходимости в добавлении блока переполнения нет. Этот процесс проиллюстрирован на рис. 3.3.

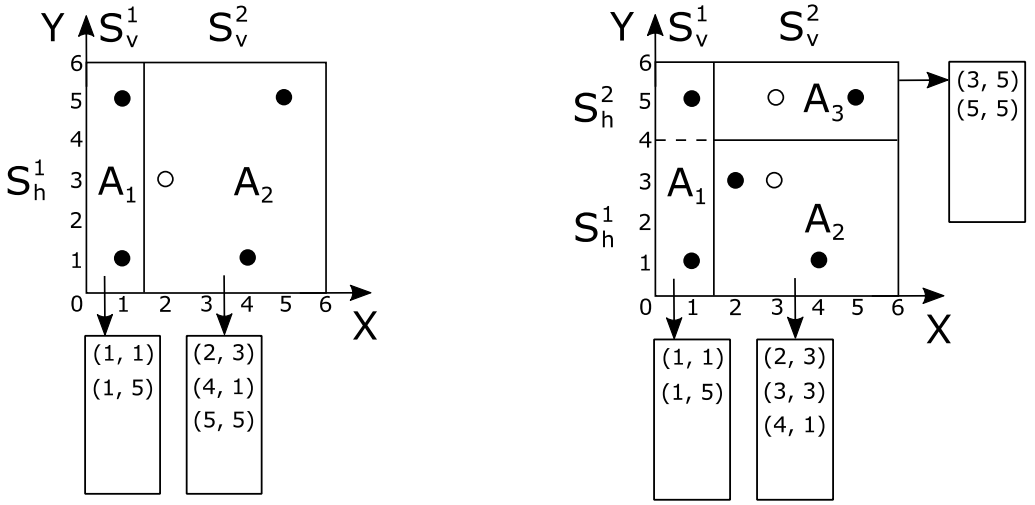


(a) Начальное состояние (b) Добавление точек (1,5) и (4,1)

Рис. 3.3. Операция вставки

Добавим еще одно значение (2; 3). При добавлении его в сегмент A_1 получим $r = 5$; $c = 1$ и $k = 5 > 4$, таким образом, нужно выполнить разделение сегмента A_1. После вертикального разделения по линии $x = 1.5$ образуются два сегмента A_1, A_2 и новая вертикальная полоса S_v^2. Теперь для полосы S_h^1 параметр c равен 2, поскольку S_h^1 теперь пересекает 2 сегмента, и $k = 2.5$ позволяет завершить операцию вставки.

При добавлении еще двух значений (3; 3) и (3; 5) возникает необходимость разделить сегмент A_2, так как $k = 5 > 4$ для полосы S_v^2. Разделим его горизонтально по прямой $y = 4$. Эта прямая также пересекает сегмент A_1, но его разделение не осуществляется. Результаты вышеприведенных операций проиллюстрированы на рис. 3.4. Воображаемое деление сегмента A_1 представлено пунктиром.



(a) Вставка (2, 3) (b) Вставка (3, 3) and (3, 5)

Рис. 3.4. Операция разделения

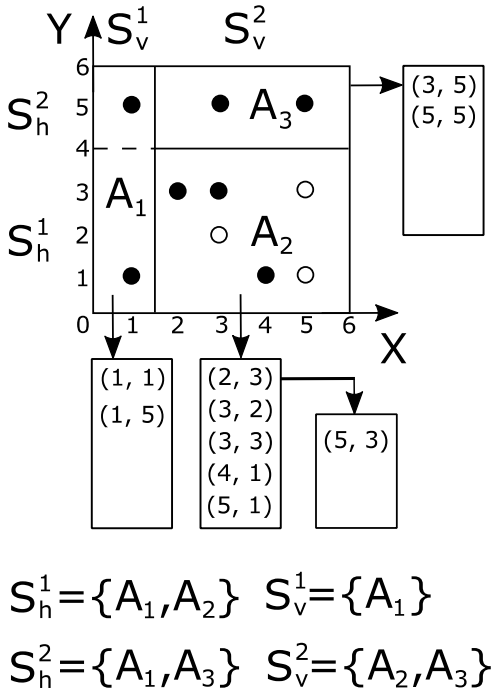


Рис. 3.5. Использование блока переполнения

Наконец, добавим еще 3 записи: (3; 2), (5; 1) и (5; 3). В сегменте A_2 достаточно свободной памяти для хранения записей (3; 2) и (5; 1), однако третья запись (5; 3) в нем не умещается, так как соответствующий блок оказывается полон. Однако, для обеих полос S_h^1 и S_v^2, пересекающих сегмент A_2, имеют место равенства $r = 8,\ c = 2$ и $k = 4$, и можно избежать разделения сегмента путем добавления блока переполнения. Результат такой вставки изображен на рис. 3.5.

Операция удаления. При удалении значения, так же как при вставке, сперва необходимо найти сегмент, которому принадлежит данное значение, загрузить необходимый блок данных и осуществить удаление. После удаления возможна ситуация, когда соответствующий сегмент становится пуст. В таком случае его можно объединить с некоторым из соседних сегментов с помощью операции слияния, описанной ниже. Операция слияния иногда возможна даже если сегмент не становится полностью пустым, при выполнении условия, что в результате в для каждой полосы среднее количество блоков переполнения для пересекаемых ею сегментах не будет

превосходить 1. Сложность операции удаления полностью совпадает со сложностью операции вставки.

Операция слияния. Пусть необходимо произвести слияние сегментов c_1 и c_2, имеющих общую границу в измерении d, в сегмент c. Для любой полосы s, если ее список указателей на сегменты содержит указатели на c_1 либо c_2, они должны быть замещены указателем на c. Сложность операции слияния оценивается аналогично сложности операции разделения.

Алгоритм слияния иллюстрирует следующий псевдокод:

Алгоритм 3.1.7. Слияние двух сегментов

```
Merge(c_1, c_2):
    in: c_1, c_2 – сегменты, подлежащие слиянию
    1:  d ← BoundaryDimension(c_1, c_2)
    2:  Загрузить первые блоки сегментов c_1, c_2 как b_1, b_2 соответственно
    4:  repeat
    3:      for all r in b_2 do
    4:          b_1 ← AddRecordToBlock(r, b_1)
    5:      end for
    6:  b_2 ← LoadNextBlock(c_2)
    7:  until Exists(b_2)
    8:  for all s ∈ StripesOfChunk(c_2) do
    9:      Добавить указатель на сегмент c_1 в полосу s
    10:     Удалить указатель на сегмент c_2 из полосы s
    11: end for
    12: return c_1
```

3.2 Альтернативная структура сеточного файла

В данном параграфе предлагается альтернативная структура сеточного файла, являющаяся дальнейшей модификацией структуры, предложенной в предыдущем параграфе, и позволяющая уменьшить размер директории индекса от $O(n^2m^2)$ до $\boldsymbol{O(n^2m)}$. Для достижения этой цели проводится реорганизация системы хранения

указателей на сегменты, позволяющая сегментам хранить указатели друг на друга. Формально, множество указателей на сегменты определяется следующим образом:

Определение 3.2.1. Пусть на множестве сегментов определено отношение полного порядка $<$. Обозначим проекцию сегмента c на измерение d через $\pi_d(c)$. Множество указателей на сегменты R определяется следующим образом:

1. Для любой пары сегментов a, b, т.ч. $a < b$ и существует измерение d, т.ч. $\pi_d(a) \subseteq \pi_d(b)$, и не существует сегмента c, т.ч. $a < c < b$ и $\pi_d(a) \subseteq \pi_d(c) \subseteq \pi_d(b)$, существует указатель $(a, b) \in R$. Назовем его *указателем в измерении* d. Данный указатель хранится в списке указателей сегмента a.
2. Для любого сегмента a и полосы s измерения d, если в R не сущетвует указателя на сегмент a в измерении d, то существует указатель $(s, a) \in R$. Данный указатель хранится в списке указателей полосы s.

Пример эквивалентных сеточных файлов, построенных согласно исходной и модифицированной структурам, приведен на рис. 3.6.

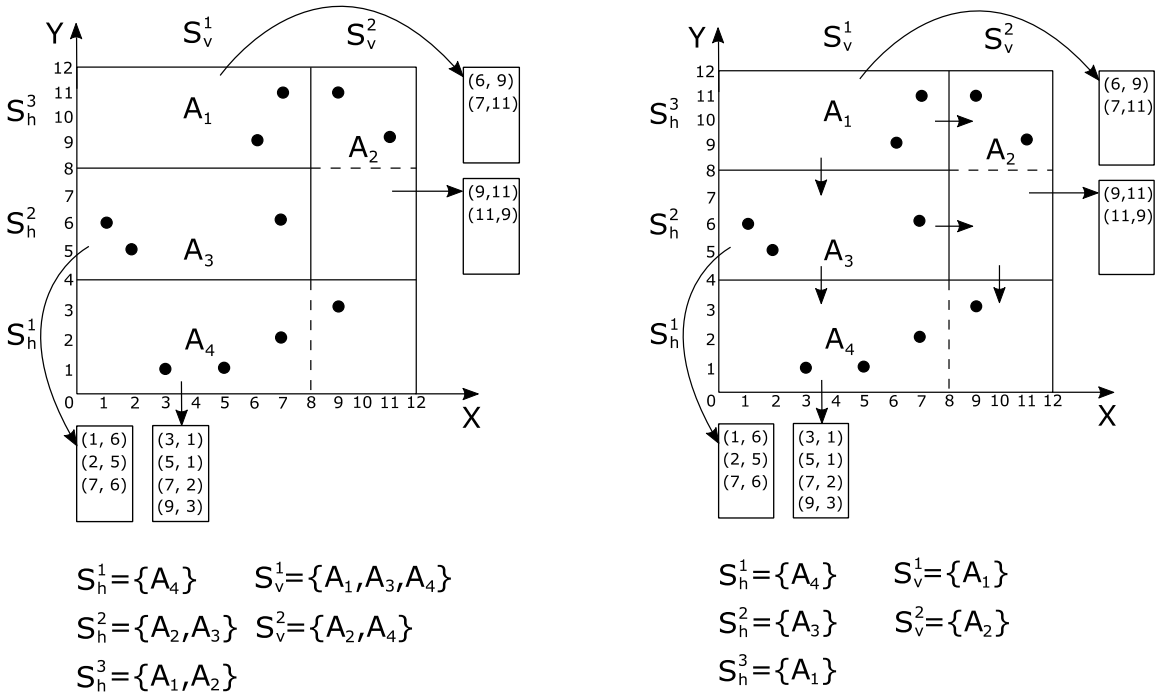


Рис. 3.6. Альтернативная структура сеточного файла

3.2.1 Оценка размера директории индекса

Для оценки размера директории индекса посчитаем общее количество хранимых указателей. Для этого, оценим по отдельности размеры множеств указателей, хранимых в списках указателей полос (указатели вида (s, a), s – полоса, a - сегмент), и множеств указателей, хранимых в списках указателей сегментов (указатели вида (a, b), a, b – сегменты).

1. Количество указателей, хранимых в списках указателей полос.

Пусть фиксирована полоса s измерения d. Рассмотрим упорядоченную последовательность сегментов $c_1, c_2, \dots, c_k$ $(c_i < c_j \Leftrightarrow i < j)$, пересекаемых полосой s. Обозначим данную последовательность как C. Для каждой пары сегментов этой последовательности c_i и c_j $(i < j)$ рассмотрим их проекции $\pi_d(c_i)$ и $\pi_d(c_j)$. Эти проекции имеют непустое пересечение, так как оба сегмента пересекаются полосой s. Существуют 4 возможные относительные позиции проекций of $\pi_d(c_i)$ и $\pi_d(c_j)$, и лишь одна из них удовлетворяет условию $\pi_d(c_i) \subseteq \pi_d(c_j)$. Без нарушения общности предполагая, что эти 4 случая имеют равную вероятность для произвольно взятой последовательной пары сегментов (c_i, c_{i+1}) рассматриваемой последовательности, можно сказать, что вероятность существования указателя из сегмента c_i в сегмент c_{i+1} есть:

$$P\big((c_i, c_{i+1}) \in R\big) = \frac{1}{4}$$

Согласно определению 3.2.1, для того, чтобы в списке указателей полосы s хранился указатель на сегмент c_j, необходимо и достаточно, чтобы не существовало сегмента $c_i < c_j$, хранящего указатель на c_j. Обозначим вероятность такого события через p_j:

$$p_j = P\big(\nexists i < j \cdot (c_i, c_j) \in R\big)$$

Введем следующее обозначение:

$$d_{i,j} = P\big(\nexists k \in [i, j-1] \cdot (c_k, c_j) \in R\big)$$

Заметим, что $p_j = d_{1,j}$. Исходя из того, что $d_{j-1,j} = 1 - P\left((c_{j-1}, c_j) \in R\right) = \frac{3}{4}$, и что $d_{i,j} = \frac{3}{4} d_{i+1,j}$ получим $d_{i,j}$=$\left(\frac{3}{4}\right)^{j-i}$.Таким образом, вероятность хранения указателя на сегмент c_j в списке указателей полосы s равна $p_j = d_{1,j} = \left(\frac{3}{4}\right)^{j-1}$.

Для подсчета общего количества указателей, хранимых в списке полосы s, введем множество индикаторных случайных величин ξ_j:

$$\xi_j = \begin{cases} 1, & (s, c_j) \in R \\ 0, & (s, c_j) \notin R \end{cases}$$

При этом $\mu(\xi_j) = P(\xi_j = 1) = p_j = \left(\frac{3}{4}\right)^{j-1}$.

Сумма данных величин равна общему количеству указателей вида (s, c_j). Посчитаем ее математическое ожидание:

$$\mu\left(\sum_{j=1}^{k} \xi_j\right) = \sum_{j=1}^{k} \mu(\xi_j) = \sum_{j=1}^{k} \left(\frac{3}{4}\right)^{j-1} = 4\left(1 - \left(\frac{3}{4}\right)^{k}\right) < 4$$

Таким образом, ожидаемое количество указателей, хранимых в списке каждой полосы, не превосходит 4. Так как общее количество полос имеет порядок $O(nm)$, то ожидаемое общее количество указателей вида (s, c_j) также будет иметь порядок $O(nm)$.

2. Количество указателей, хранимых в списках указателей сегментов.

Сперва зафиксируем некоторое измерение d и посчитаем количество подобных указателей в нем. Рассмотрим упорядоченную последовательность всех сегментов $c_1, c_2, \dots, c_k$ $(c_i < c_j \Leftrightarrow i < j)$. Обозначим через p вероятность того, что для произвольно взятой пары сегментов c_i, c_j $(i < j)$ их проекции на измерение d удовлетворяют условию $\pi_d(c_i) \subseteq \pi_d(c_j)$. Обозначим через q вероятность того, что для произвольно взятой тройки сегментов c_i, c_k, c_j $(i < k < j)$, их проекции на измерение d удовлетворяют условию $\pi_d(c_i) \subseteq \pi_d(c_k) \subseteq \pi_d(c_j)$. Заметим, что $p \geq q$.

Оценим вероятность существования указателя из сегмента c_i в сегмент c_j для произвольной пары сегментов (c_i, c_j) рассматриваемой последовательности. Такой

указатель существует, согласно определению 3.2.1, если единовременно выполняются оба следующих условия:

1. $\pi_d(c_i) \subseteq \pi_d(c_j)$
2. $\nexists k \in [i+1, j-1]$ т.ч. $\pi_d(c_i) \subseteq \pi_d(c_k) \subseteq \pi_d(c_j)$

Рассмотрим произвольный индекс $k \in [i+1, j-1]$ и оценим вероятность события $\pi_d(c_i) \subseteq \pi_d(c_k) \subseteq \pi_d(c_j)$. Обозначим данное событие через A_k. Обозначим событие $\pi_d(c_i) \subseteq \pi_d(c_j)$ через B и посчитаем вероятность того, что произошло событие A_k, при условии, что произошло событие B. Заметим, что событие A_k влечет за собой событие B, т.е. $P(B|A_k) = 1$. Используя формулу Байеса, получим:

$$P(A_k|B) = \frac{P(B|A_k)P(A_k)}{P(B)} = \frac{P(A_k)}{P(B)} = \frac{q}{p}$$

Учитывая, что события $\{A_k\}$ независимы, получим, что

$$P\left((c_i, c_j) \in R\right) =$$

$$P\left(\pi_d(c_i) \subseteq \pi_d(c_j) \wedge \forall k \in [i+1, j-1] \cdot \left(\neg\left(\pi_d(c_i) \subseteq \pi_d(c_k) \subseteq \pi_d(c_j)\right)\right)\right) =$$

$$p \cdot \left(1 - \frac{q}{p}\right)^{j-i-1}$$

Для подсчета суммарного количества указателей вида (c_i, c_j), введем следующие индикаторные случайные величины:

$$\eta_{ij} = \begin{cases} 1, & (c_i, c_j) \in R \\ 0, & (c_i, c_j) \notin R \end{cases}$$

При этом $\mu(\eta_{ij}) = P(\eta_{ij} = 1) = p \cdot \left(1 - \frac{q}{p}\right)^{j-i-1}$.

Сумма данных величин равна суммарному количеству указателей вида (c_i, c_j). Посчитаем ее математическое ожидание:

$$\mu\left(\sum_{i=1}^{k-1}\sum_{j=i+1}^{k}\eta_{ij}\right) = \sum_{i=1}^{k-1}\sum_{j=i+1}^{k}\mu(\eta_{ij}) = \sum_{i=1}^{k-1}\sum_{j=i+1}^{k} p \cdot \left(1 - \frac{q}{p}\right)^{j-i-1}$$

$$= \frac{p^2\left(kq + p\left(\left(1 - \frac{q}{p}\right)^k - 1\right)\right)}{q^2}$$

Так как $p \geq q$, имеет место неравенство $p\left(\left(1-\frac{q}{p}\right)^k - 1\right) \leq 0$. Учитывая, что общее количество сегментов k имеет порядок $O(nm)$, получим, что данное математическое ожидание, равное количеству указателей между сегментами для фиксированного измерения d, также имеет порядок $O(nm)$. Так как в предлагаемой структуре все n измерений рассматриваются как симметричные, ожидаемое общее количество указателей вида (c_i, c_j) имеет порядок $O(n^2 m)$.

Таким образом, ожидаемое количество всех хранимых указателей – а также и величина директории индекса – имеет порядок:

$$\boldsymbol{O(nm + n^2 m) = O(n^2 m)}.$$

3.2.2 Сравнение размера директории индекса

В данном подпараграфе приводится сравнение полученной оценки размера директории индекса с двумя методами организации сеточных файлов, описанными в [85], - многомерным динамическим хешированием и многомерным расширяемым хешированием. Оценки размеров директории для обоих случаев также приводятся в [85]: $\boldsymbol{O\left(r^{1+\frac{1}{s}}\right)}$ и $\boldsymbol{O\left(r^{1+\frac{n-1}{ns-1}}\right)}$ соответственно, где r есть количество записей, s – размер блока (выраженный в среднем количестве записей, которое можно сохранить в блоке), а n – количество измерений. Необходимо заметить, что рассматривается случай равномерного распределения данных [44], [85].

Для осуществления сравнения выразим размер директории индекса в предлагаемом в подходе через данные величины. Так как каждый сегмент имеет не более одного блока переполнения в среднем, можно без нарушения общности предположить, что каждый сегмент будет хранить в среднем $\frac{3}{2}s$ значений. Таким образом можно заключить, что для хранения всех r записей будет необходимо в среднем $\frac{2r}{3s}$ сегментов, что имеет порядок $O(nm)$. Таким образом, согласно §3.2.1, размер директории индекса в предлагаемом подходе может быть оценен как

$O(n^2m) = O\left(\frac{nr}{s}\right)$. По сравнению с подходами MDH и MEH, размер директории индекса в предлагаемом подходе меньше в $\frac{sr^{\frac{1}{s}}}{n}$ и $\frac{sr^{\frac{n-1}{ns-1}}}{n}$ раз соответственно.

3.2.3 Операции над сеточными файлами

Операции поиска, вставки и удаления. Существенным различием от структуры, рассмотренной в §3.1, при осуществлении операций поиска, вставки или удаления является то, что множества пересекаемых полосой сегментов не хранятся отдельно для каждой полосы. Для того, чтобы посетить пересекаемые данной полосой s измерения d сегменты, необходимо рассмотреть все сегменты, указатели на которые хранятся в списке указателей полосы s, и для каждого из них проитерировать по хранящимся в них указателям измерения d. Это делает затруднительным использование алгоритмов быстрого пересечения множеств, так как искомые множества не могут быть заранее представлены в виде необходимых структур. Оценки данных операций без применения алгоритмов быстрого пересечения множеств и количества требуемых дисковых операций приведены в §3.1.2.

Операции разделения и слияния. Пусть необходимо осуществить разделение сегмента c на сегменты c_1 и c_2 по измерению d. Некоторая полоса s измерения d при этом разделяется на полосы s_1 и s_2. При этом необходимо совершить количество операций, пропорциональное суммарному количеству указателей, хранящихся в списках указателей рассматриваемых полос и сегментов. Количество указателей, хранящихся в списке указателей полосы, как указано выше, не превосходит 4 в среднем. Среднее количество указателей, хранящихся в списке каждого сегмента, имеет порядок $O(n)$, т.к. общее количество таких указателей среди $O(nm)$ сегментов имеет порядок $O(n^2m)$. Суммарное количество указателей на конкретный сегмент может быть оценено как отношение общего количества указателей к общему количеству сегментов. Общее количество указателей в среднем есть $4nm + n^2m$ – по 4 указателя в списке каждой из nm полос, и по n указателей в списке каждого из nm сегментов. Таким образом количество указателей на сегмент имеет порядок

$$O\left(\frac{4nm + n^2m}{nm}\right) = O(n).$$

Таким образом, сложность операции разделения также составляет $\boldsymbol{O(n)}$. Оценка количества дисковых операций аналогична оценке, приведенной в §3.1.2. Операция слияния имеет аналогичную сложность.

Пример операции разделения изображен на рис. 3.7.

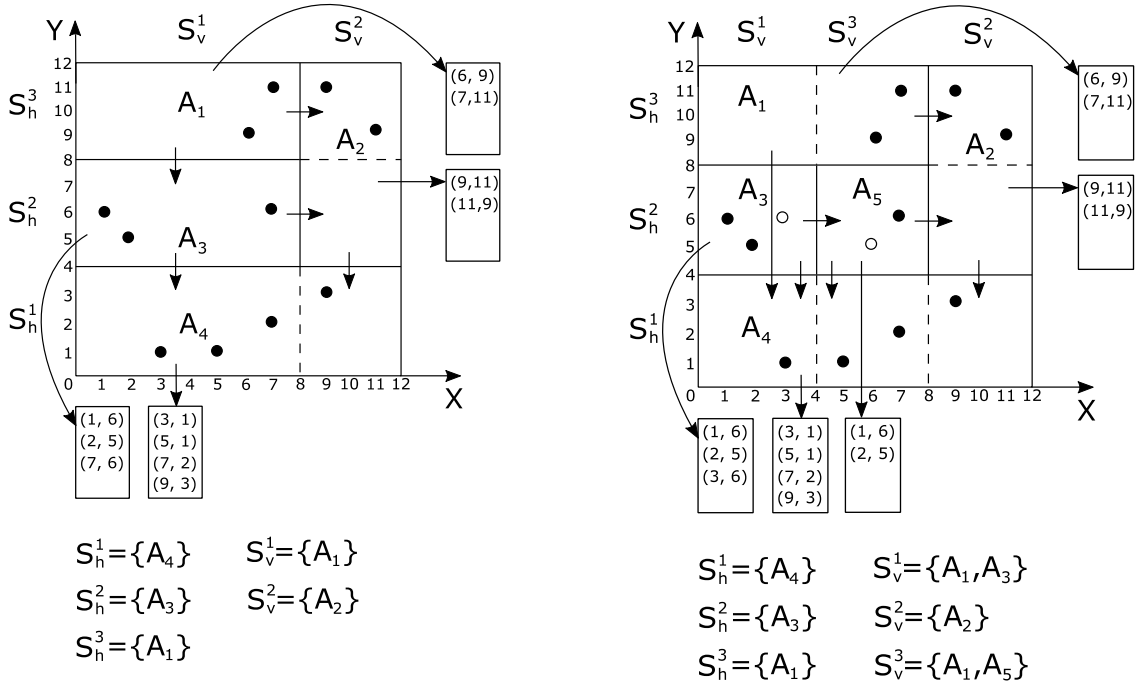


Рис. 3.7. Операция разделения

3.2.4 Представление сеточного файла в виде ориентированного ациклического графа

Предлагаемая динамическая структура индекса позволяет естественным образом представить сеточный файл в виде ориентированного ациклического графа. Подобное представление используется в качестве реализации индекса.

Определение 3.2.2. Скажем, что граф $G = \langle V, E \rangle$ представляет сеточный файл $F = \langle D, S, C \rangle$, имеющий множество указателей R, если он построен согласно следующим условиям:

1. Для каждого сегмента $a \in C$ существует соответствующая ему вершина $v_a \in V$;
2. Для каждой полосы $s \in S$ существует соответствующая ей вершина $v_s \in V$.
3. Для каждого указателя $(a, b) \in R$ существует ребро $(v_a, v_b) \in E$, где a – полоса либо сегмент, b – сегмент.

На рис. 3.8. приведен пример графа, соответствующего сеточному файлу, изображенному на рис. 3.7.

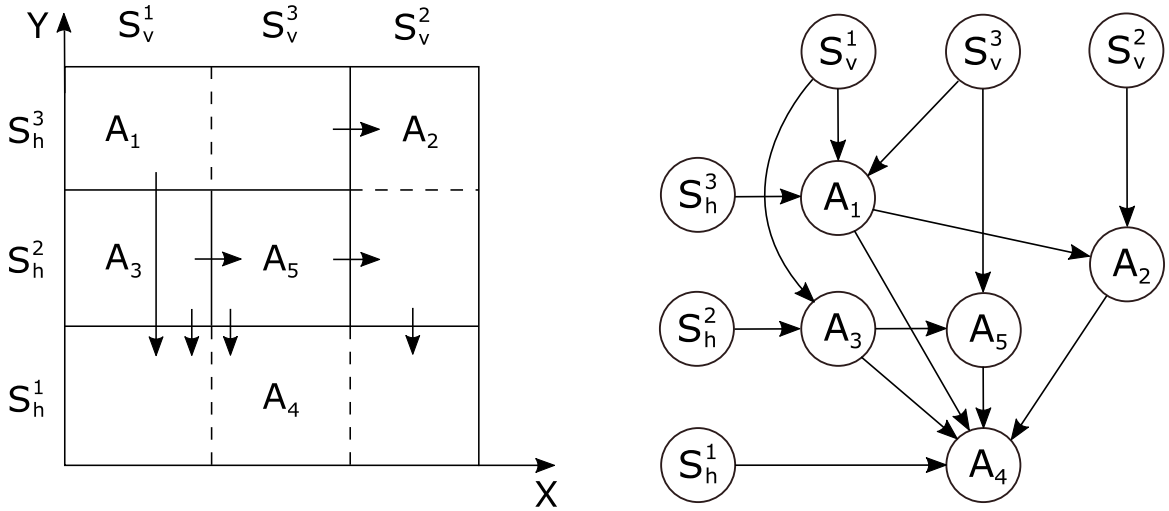


Рис. 3.8. Представление сеточного файла в виде графа

3.3 Формализация концепции сеточного файла с помощью XML

Целью XML-формализации концепции сеточного файла является создание языка определения директории, независимого от парадигм управления данными. Формализация концепции сеточного файла посредством XML предполагает разработку XML-приложения. В этом приложении понятия измерения, полосы и сегмента сеточного файла определяются как XML-элементы. XML-представление концепции сеточного файла определяется с помощью DTD, приведенной в §2.2. Элемент *STRIPE* является пустым элементом и описывается с помощью пяти атрибутов: *ref_to_chunk, min_val,*

max_val, *rec_cnt* и *chunk_cnt*. Значения атрибутов *ref_to_chunk* являются указателями на пересекаемые полосой сегменты. С помощью атрибутов *min_val* (нижняя граница) и *max_val* (верхняя граница) определяются границы полосы. При этом точки, лежащие на нижней границе, считаются принадлежащими полосе, а на верхней границе – нет. Значения атрибутов *rec_cnt* и *chunk_cnt* определяют, соответственно, общее количество записей в полосе и количество пересекаемых ею сегментов. Содержимое элемента *CHUNK* зависит от атрибутов *id* типа *ID*, *avg*, *ref_to_db* и *ref_to_chunk*. Атрибут *ref_to_db* содержит указатель на соответствующий сегменту блок данных. Атрибуты *ref_to_chunk* используются для хранения указателей на другие сегменты. Значения атрибутов *avg* используются во время реорганизации сеточного файла и содержат средние значения координат записей, принадлежащих сегменту, по каждому измерению. Содержание элемента *DIM* зависит от внутренних элементов *STRIPE* и имеет единственный атрибут *name* – наименование измерения. На рис. 3.9 приведен пример представления в виде графа XML-схемы, удовлетворяющей данной DTD.

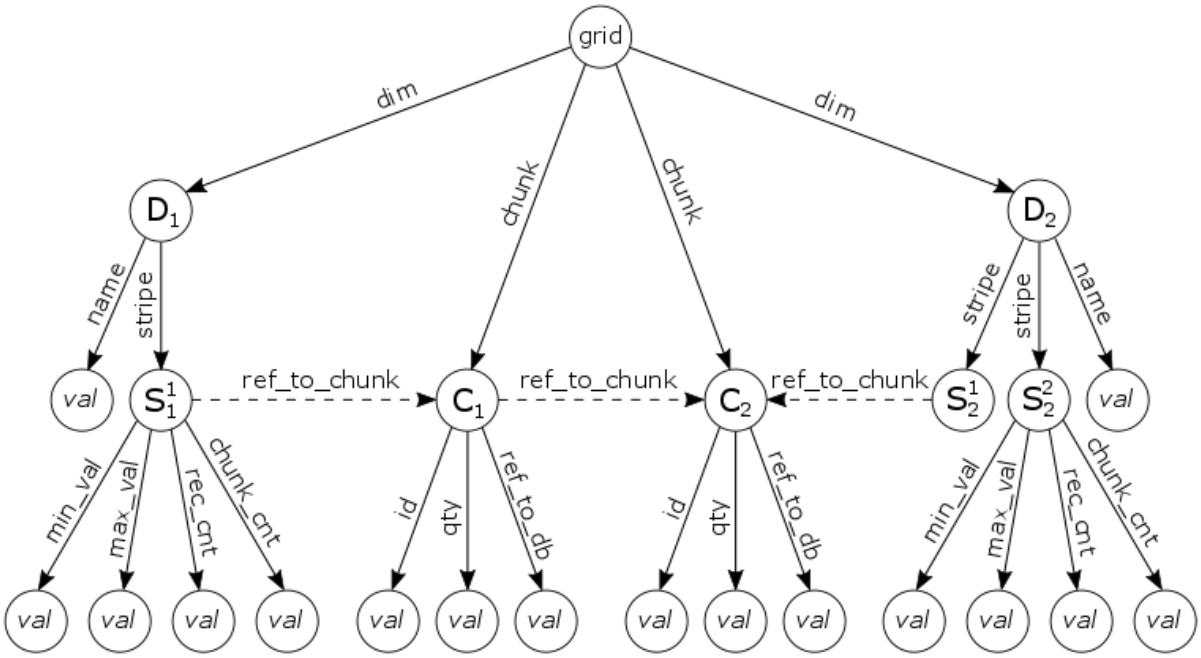


Рис. 3.9. Граф XML DTD

Метки ребер графа на этом рисунке имеют две функции, объединяющие информацию, содержащуюся в элементах и объявлениях отношений. Пусть ребро, ведущее из вершины *N* в вершину *M*, имеет метку *L*.

1. Вершина *N* может быть рассмотрена как объект или структура, а вершина *M* как один из атрибутов объекта или поле структуры. В этом случае *L* является именем атрибута или поля соответственно.
2. Вершины *N* и *M* могут быть рассмотрены как объекты, а *L* – как отношение между ними.

3.4 Выводы по главе

В главе 3 разработан подход к построению хранилища данных для канонической модели, в основном ориентированный на поддержку OLAP приложений. Предложена динамическая структура индекса для многомерных данных, эффективные алгоритмы для ее поддержки и приведены оценки сложности предложенных алгоритмов. В качестве формализма для управления многомерными данными использована концепция сеточных файлов.

- Предложена модификация структуры сеточного файла, имеющая целью исключить хранение повторяющихся указателей на одни и те же блоки данных, а также обеспечить плавный рост размера директории индекса и разумные стоимости операций;
- Приведены характеристики модифицированной структуры сеточных файлов и оценки их величин;
- Приведены операции над сеточными файлами, адаптированные к применению над предложенной модифицированной структурой, и оценки их сложности;
- Проведено сравнение полученной оценки размера директории индекса с двумя методами организации сеточных файлов – многомерным динамическим хешированием и многомерным расширяемым хешированием;
- Предложен способ представления сеточного файла в виде ориентированного ациклического графа, используемого в качестве реализации индекса;
- Предложена формализация концепции сеточного файла с помощью XML.

4 РАЗРАБОТКА ПРОГРАММНОЙ РЕАЛИЗАЦИИ И ЭКСПЕРИМЕНТАЛЬНЫЕ ИССЛЕДОВАНИЯ

На основе предложенной в главе 3 схеме динамического индекса для многомерных данных был реализован прототип хранилища данных. Для создания программного обеспечения был использован язык C++. Был проведен ряд экспериментов для сравнения производительности построенного прототипа с MongoDB [43]. MongoDB была выбрана для сравнения по прагматическим соображениям, т.к. является на сегодняшний день одной из наиболее востребованных NoSQL баз данных. Далее приводятся архитектура разработанного программного обеспечения и результаты экспериментальных исследований.

4.1 Разработка и реализация прототипа хранилища данных

Программа написана на языке C++. Целью программы является реализация структуры динамического индекса, предложенной в главе 3, для получения экспериментальных данных о ее работе. На рис. 4.1 приведена UML-диаграмма классов разработанной программы.

Далее приводится детальное описание используемых классов с информацией об их функциональности, предоставляемом интерфейсе и используемых алгоритмах и структурах данных.

`class Value.` Представляет собой вспомогательную структуру, реализующую точку в n-мерном пространстве. Хранит в себе вектор из n целочисленных координат точки, представленных в виде 32-битных беззнаковых целых чисел.

`class Record.` Используется для представления и работы с записями, инкапсулирует информацию о формате хранения записи на диске и предоставляет следующий интерфейс:

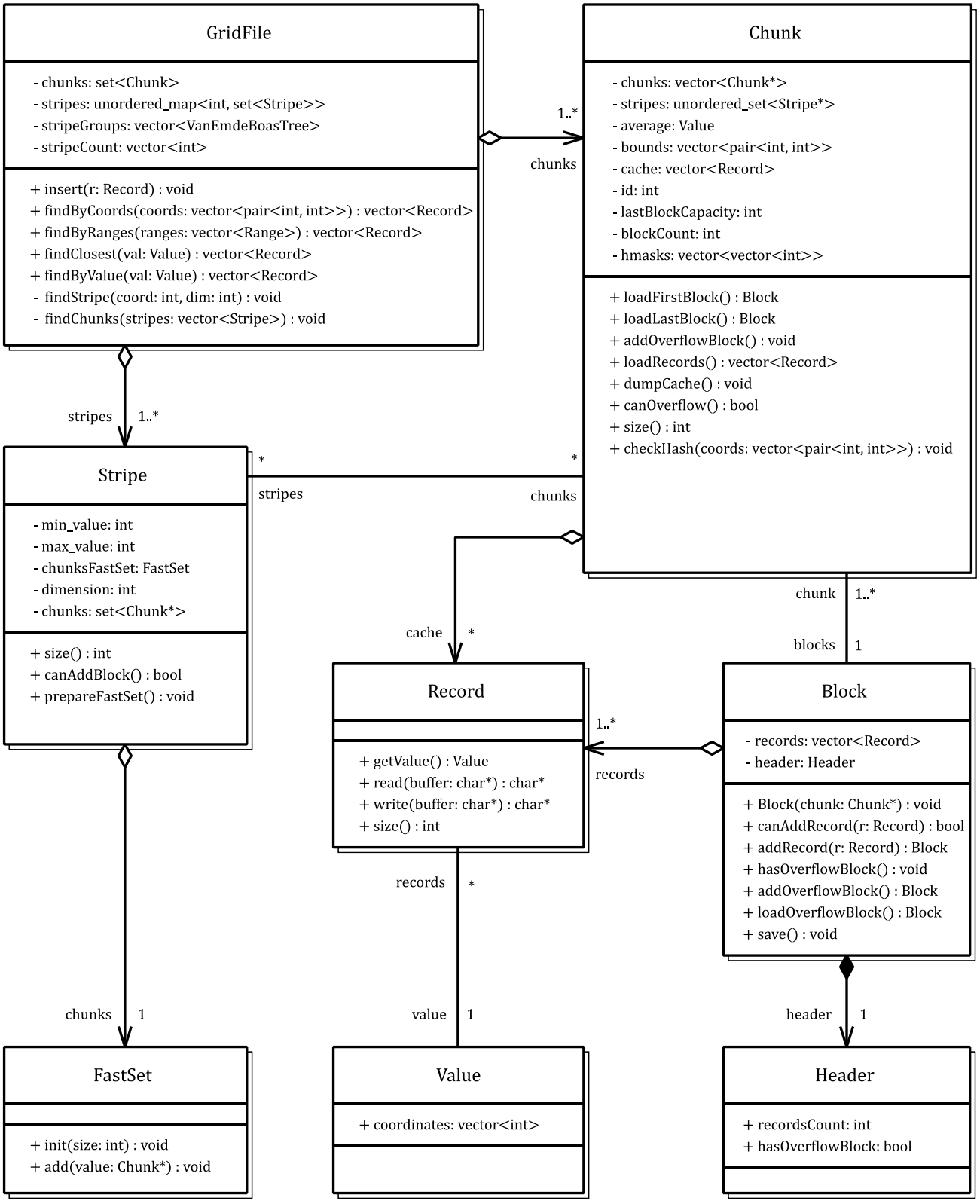


Рис. 4.1. UML диаграмма классов разработанной программы

- `getValue()`. Возвращает точку n-мерного пространства, соответствующую данной записи. Для подсчета значений координат используются хеш-функции. Набор хеш-функций заранее определен и является единым для всех записей.

- `read(in: buffer)`. Производит чтение записи из заданного буфера данных. Буфер представляет собой указатель на участок памяти, начиная с которого нужно осуществить чтение. Данные при вводе/выводе сериализуются в бинарный формат. При работе с записями, поля которых имеют значения нефиксированной длины, такими как строки, для каждой записи хранится два поля – длина и значение.

- `write(in: buffer)`. Записывает значение в предоставленный буфер памяти. Для каждого поля структуры *record* последовательно записываются бинарные представление длины этого поля и его значение. В случаях, когда значения полей структуры имеют фиксированную длину, в целях экономии памяти длина поля не записывается. Формат хранения записи на диске приведен на рис. 4.2.

4 байт	L_1 байт	4 байт	L_2 байт	…	4 байт	L_n байт
Длина поля 1 (L_1)	Поле 1	Длина поля 2 (L_2)	Поле 2		Длина поля 2 (L_n)	Поле n

Рис. 4.2. Формат хранения записи

- `size()`. Возвращает объем памяти, занимаемый значением будучи записанным на диске.

Таким образом, в данном классе инкапсулируется вся логика, связанная с форматом представления значений для хранения на диске и соответствующими операциями ввода/вывода.

`class Block`. Представляет собой структуру для работы с блоками данных. Инкапсулирует внутреннее представление блока данных, позволят осуществлять операции чтения/записи блоков а также манипулировать блоками переполнения.

- `Block(in: chunk, index)`. Конструктор класса, производит чтение блока заданного номера для данного сегмента с диска и инициализирует поля класса.

- `canAddRecord(in: record)`. Булева функция, проверяющая, можно ли добавить заданное значение в блок.

- `addRecord(in: record)`. Добавляет заданное значение в блок. В случае недостатка свободного места создает блок переполнения и записывает значение в него. Проверка возможности создания блока переполнения с точки зрения структуры сеточного файла должна осуществляться пользователем предварительно.

- `hasOverflowBlock()`. Булева функция, отвечающая на вопрос «есть ли у данного блока блок переполнения». Использует информацию, хранящуюся в заголовке блока.

- `addOverflowBlock()`. Создает пустой блок переполнения и возвращает соответствующий ему объект класса Block. В заголовке текущего блока отмечается наличие блока переполнения.

- `loadOverflowBlock()`. Загружает блок переполнения данного блока. В случае отсутствия такового генерирует исключение. Проверка на существование блока переполнения должна проводиться пользователем предварительно.

- `save()`. Сохраняет блок на диске. При сохранении создается буфер данных, в который сперва записывается заголовок блока, содержащий информацию о количестве хранимых записей и наличии блока переполнения, затем вызывается операция записей для каждого из хранимых в блоке значений. После этого осуществляется запись буфера на диск в соответствующий файл. Формат хранения на диске блока данных, содержащего k записей объемов $R_1, R_2 \ldots, R_k$, представлен на рис. 4.3.

1 байт	4 байт	R_1 байт	R_2 байт	…	R_k байт
Флаг Б/П	Количество записей (k)	Запись 1	Запись 2		Запись k

Рис. 4.3. Формат хранения блока данных

Данный класс хранит в себе вектор значений, содержащихся в блоке, и его заголовок. Заголовок блока является объектом внутреннего класса `header` и имеет поля `hasOverflowBlock` и `numberOfRecords`.

`class Stripe.` Реализует полосу сеточного файла. Хранит информацию о границах полосы, измерении, которому она принадлежит, и пересекаемых ею сегментах.

- `canAddBlock().` Булева функция, осуществляющая проверку на возможность добавления блока переполнения в один из пересекаемых ею сегментов. Инкапсулирует логику работы с полосой как с линейной хеш-функцией, вычисляя среднее количество блоков переполнения среди пересекаемых данной полосой сегментов и сравнивая ее с единицей.

- `prepareFastSet().` Представляет множество пересекаемых полосой сегментов в виде множества, подготовленного для использования алгоритмов быстрого пересечения множеств, и сохраняет его как значение своего поля `fastSet`.

- `size().` Возвращает количество пересекаемых данной полосой сегментов.

`class Chunk.` Представляет собой реализацию сегмента сеточного файла. Предоставляет интерфейс для работы с блоками данных, соответствующими данному сегменту, а также для добавления новых значений и чтения существующих. В целях улучшения производительности, каждый сегмент имеет собственных кеш значений в виде буфера, в который данные записываются при добавлении в сегмент. Когда кеш заполняется, либо когда возникает необходимость разделения сегмента, данные, хранящиеся в кеше, переносятся на диск. Таким образом значительно сокращается количество дисковых операций при добавлении записей. Ниже приведен интерфейс класса с описанием используемых алгоритмов.

- `loadFirstBlock().` Загружает с диска первый блок сегмента.

- `loadLastBlock().` Загружает с диска последний блок сегмента.

- `addOverflowBlock().` Добавляет блок переполнения к данному сегменту.

- `loadRecords().` Производит последовательное чтение блоков сегмента и возвращает вектор всех хранящихся в них записей. Используется при поиске данных.

- `dumpCache()`. Сохраняет кэш в последнем блоке, соответствующем сегменту. Размер кэша регулируется в процессе работы и поддерживается на таком уровне, чтоб не превышать объем свободной памяти в последнем блоке переполнения сегмента. Таким образом, запись кэша на диск в любой момент времени потребует одну дисковую операция.

- `canOverflow()`. Проверяет возможность добавления блока переполнения к сегменту. Для осуществления проверки вызываются функции `canAddBlock()` каждой пересекаемой сегментом полосы. В случае, если хотя бы для одной полосы добавить новый блок невозможно, функция возвращает false. В противном случае возвращает true.

- `checkHash(coords: in)`. Осуществляет проверку заданных координат на несоответствие данным, хранящимся в сегменте. Вычисляет значения множества хеш-функций `hashes` над заданными координатами, и сравнивает их с хранящимися значениями.

- `size()`. Возвращает объем памяти, занимаемый данным объектом для представления сегмента. Используется при оценке размера структуры индекса.

`class GridFile`. Основной компонент программного обеспечения, реализующий саму структуру индекса и предоставляющий интерфейс для манипуляции данными хранилища, реализующий операции вставки, удаления и поиска. Операции реализованы согласно описаниям, приведенным в главе 3.

- `Insert(in: record)`. Осуществляет вставку значения. Для заданного значения вычисляются координаты соответствующей точки с помощью функции `record.getValue()`, затем для каждой из координат вызывается функция `findStripe()` для нахождения соответствующей полосы. После этого, с помощью функции `findChunks()` находится соответствующий точке сегмент, и данное значение добавляется в него.

- `findByCoords(in: coords)`. Осуществляет поиск значений по набору заданных координат. Процедура нахождения полос, соответствующих заданным координатам, и сегментов, пересекаемых ими, аналогична описанной для операции

вставки. Поиск по заданным координатам позволяет использовать алгоритм быстрого пересечения множеств сегментов при использовании функции `findChunks()`. После получения множества сегментов, для каждого из них вызывается функция `checkHash(coords)` для проверки возможности исключения данного сегмента из рассмотрения. В случае отрицательного результата проверки осуществляется чтение записей сегмента и сравнение их с заданным набором координат.

- `findByValue(in: record)`. Реализует поиск заданного значения. Является частным случаем функции поиска по набору заданных координат и использует ее для реализации.

- `findByRanges(in: ranges)`. Осуществляет поиск в заданных диапазонах значений. Реализация аналогична функции findByCoords, за исключением того, что алгоритм быстрого пересечения множеств при вызове findChunks не используется.

- `findClosest(in: record)`. Осуществляет поиск ближайших значений заданной точки. Для этого сперва вызывается процедура поиска по значению и проверяются значения, хранящиеся в соответствующем сегменте. В случаях, когда расстояние до границы некоторых соседних сегментов меньше текущего наименьшего значения, осуществляется проверка среди значений этих сегментов также.

Ниже приводятся закрытые функции-члены класса **`GridFile`**, реализующие алгоритмы поиска полосы и нахождения пересечения множества полос. Эти функции реализуют алгоритмы, описанные в главе 3, и являются ключевыми факторами в быстродействии данной программы.

- `findStripe(in: coord, dim)`. Для нахождения полосы заданного измерения по заданной координате в данной реализации используются деревья ван Эмде Боаса. Библиотека [92] предоставляет реализацию деревьев ван Эмде Боаса в виде класса VanEmdeBoasTree для работы с диапазонами целочисленных беззнаковых чисел произвольных интервалов. В данной реализации деревья ван Эмде Боаса используются для нахождения полосы по заданной координате следующим образом. Полосы каждого измерения группируются по значениям 16 старших бит своих нижних границ. Таким образом, внутри каждой группы хранятся полосы, различающиеся только своими 16

младшими битами. На множестве 16 старших бит строится дерево ван Эмде Боаса, которое используется для нахождения номера группы искомой полосы за время $\log\log 2^{16}$. На рис. 4.4 представлена используемая схема применения дерева ван Эмде Боаса для нахождения полосы.

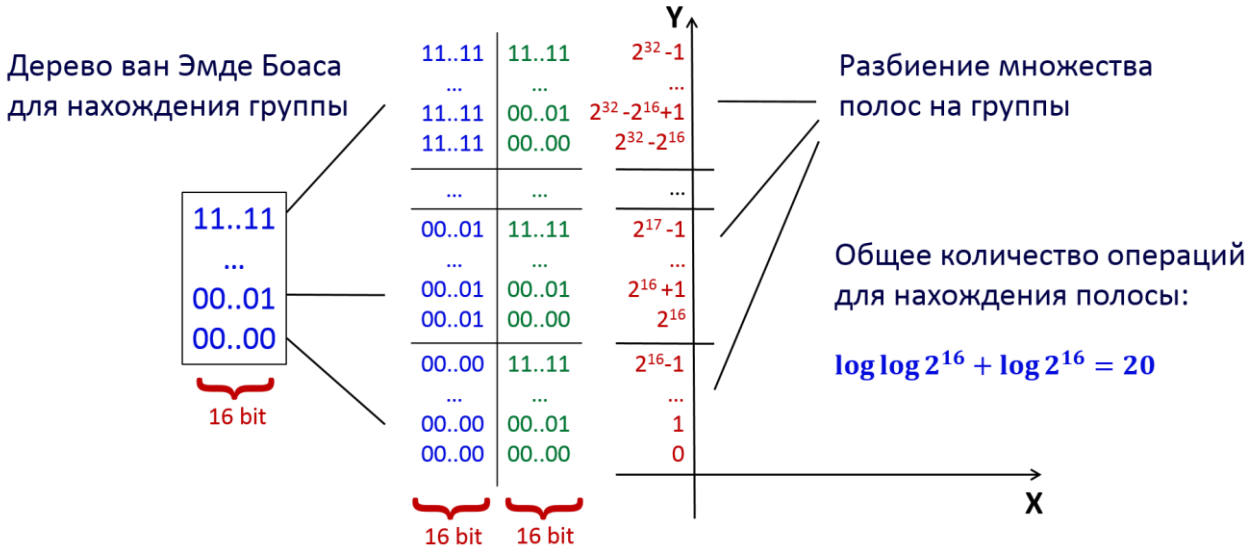


Рис. 4.4. Схема использования дерева ван Эмде Боаса для нахождения полосы

После того, как группа найдена, внутри данной группы для нахождения самой полосы осуществляется бинарный поиск. Так как внутри группы значимыми являются младшие 16 бит, бинарный поиск требует 16 операций в худшем случае. Таким образом, нахождения полосы занимает не более 20 операций.

- `findChunks(in: stripes)`. Вычисляет множество сегментов, пересекаемых каждой из множества заданных полос. В случае, когда каждому измерению соответствует не более одной полосы, для осуществления пересечения над заранее подготовленными элементами `fastSet` каждой из полос используется алгоритм быстрого пересечения множеств. Если же некоторое измерение представлено более чем одной полосой, используется следующий алгоритм. Выбирается измерение с наименьшим ненулевым количеством полос среди данного множества. Для каждой из полос этого множества проводится итерация по множеству хранимых в ней указателей на пересекаемые сегменты. Для каждого из этих сегментов и для каждого измерения

запроса проверяется, пересекается ли он какой-либо полосой данного измерения. Проверка на пересечения сегмента и полосы осуществляется путем сравнения их границ. В случае положительного результата проверки для каждого измерения, данный сегмент добавляется к ответу.

4.2 Экспериментальные исследования

Для осуществления экспериментов рассмотрены точки в трехмерном евклидовом пространстве. Координаты точек представлены в виде 32-битных беззнаковых целых чисел, таким образом, размер одной записи равен 12 байт, и в одном блоке данных объема 4096 байт может быть сохранено до 340 записей. Далее приведены результаты тестирования для операций вставки и основных категорий запросов поиска, которые сравнены с результатами аналогичных операций, проведенных над MongoDB. Для представления точек в трехмерном пространстве использовалась коллекция `grid`, документы которой имели поля “x”, “y” и “z”. По каждому из данных полей были созданы индексы. MongoDB использует пересечение индексов[1] для обработки запросов по множеству полей.

Тестирование было осуществлено с использованием операций вставки, а также четырех основных категорий запросов – поиск заданной точки, запросы к данным с совпадением отдельных координат, запросы в диапазонах значений, поиск ближайших соседних объектов [10]. Было проведено сравнение объемов памяти, используемых метаданными. Далее приведены результаты проведенного тестирования для каждой из данных категорий.

Вставка значений. На рис. 4.5 приводится сравнение быстродействия построенного прототипа (в секундах) и MongoDB при осуществлении 2 млн. операций вставки. Построенный прототип хранилища уступает MongoDB в производительности при вставке значений, причем с увеличением объема базы данных преимущество MongoDB возрастает.

[1] https://docs.mongodb.com/manual/core/index-intersection/

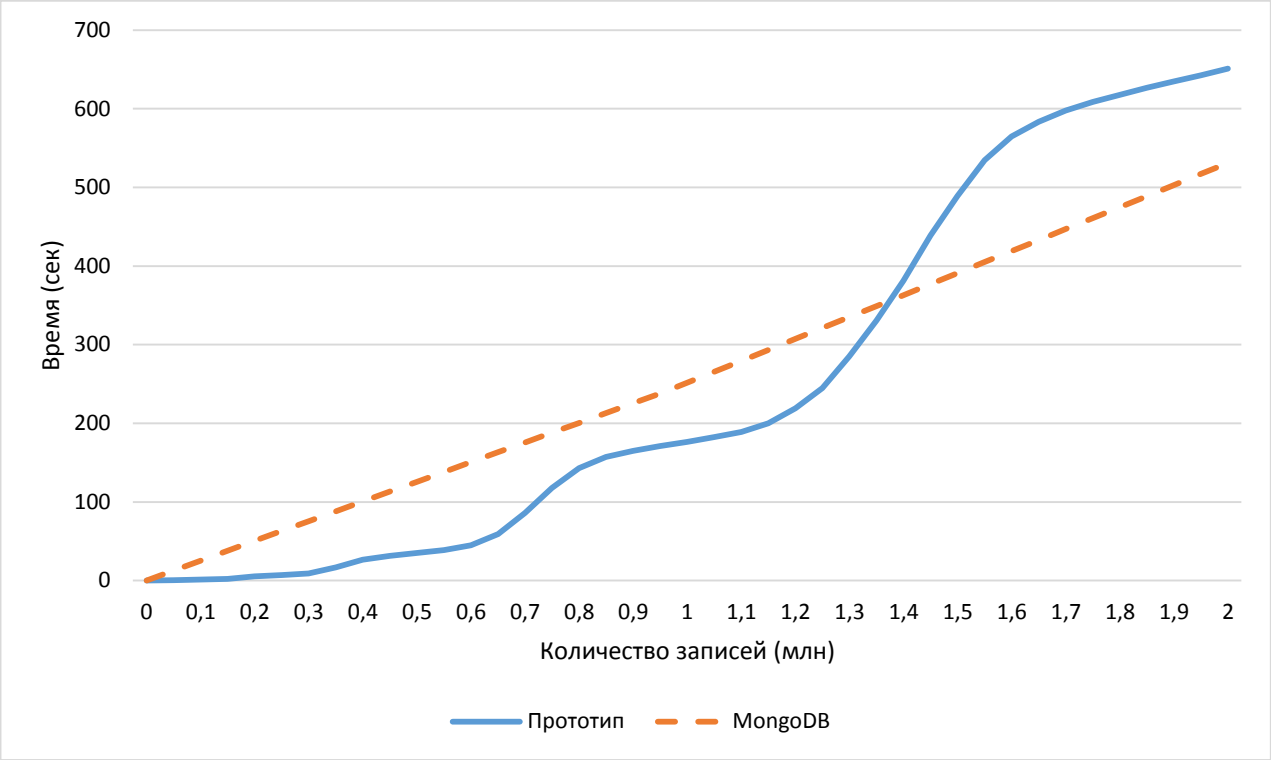


Рис. 4.5. Операция вставки

Сравнение размера директории индекса. На рис. 4.6. представлены графики роста размера директории индекса при осуществлении 2 млн. операций вставки. Ось абсцисс представляет количество хранимых в базе данных значений, а ось ординат – расходуемая индексом память (в МБ). Можно видеть, что предлагаемая динамическая структура индекса занимает гораздо меньше памяти, чем используемые MongoDB B-деревья.

Поиск заданной точки. На рис. 4.7 представлена зависимость времени обработки запросов поиска заданной точки в зависимости от количества хранимых в базе данных значений. В этом и последующих графиках рассматривается время выполнения 10^5 случайно сгенерированных запросов определенного типа. Установлено, что построенный прототип хранилища данных обладает лучшей производительностью при поиске заданной точки, чем MongoDB.

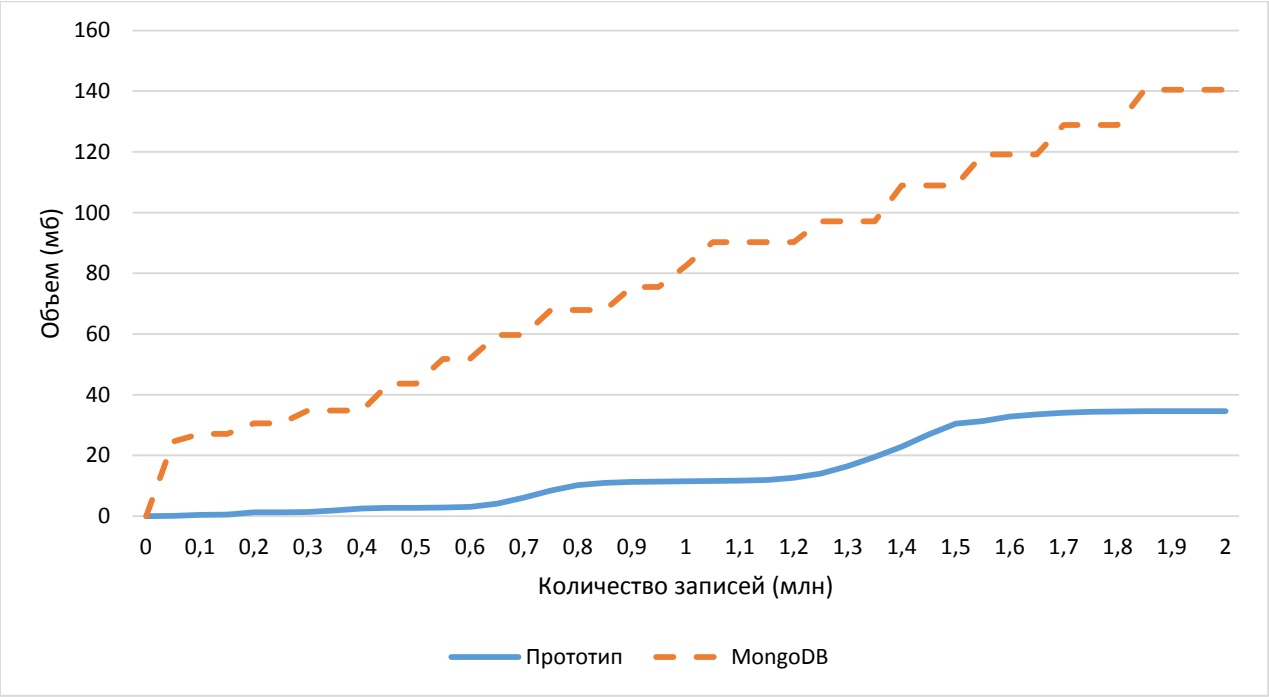


Рис. 4.6. Размер директории индекса

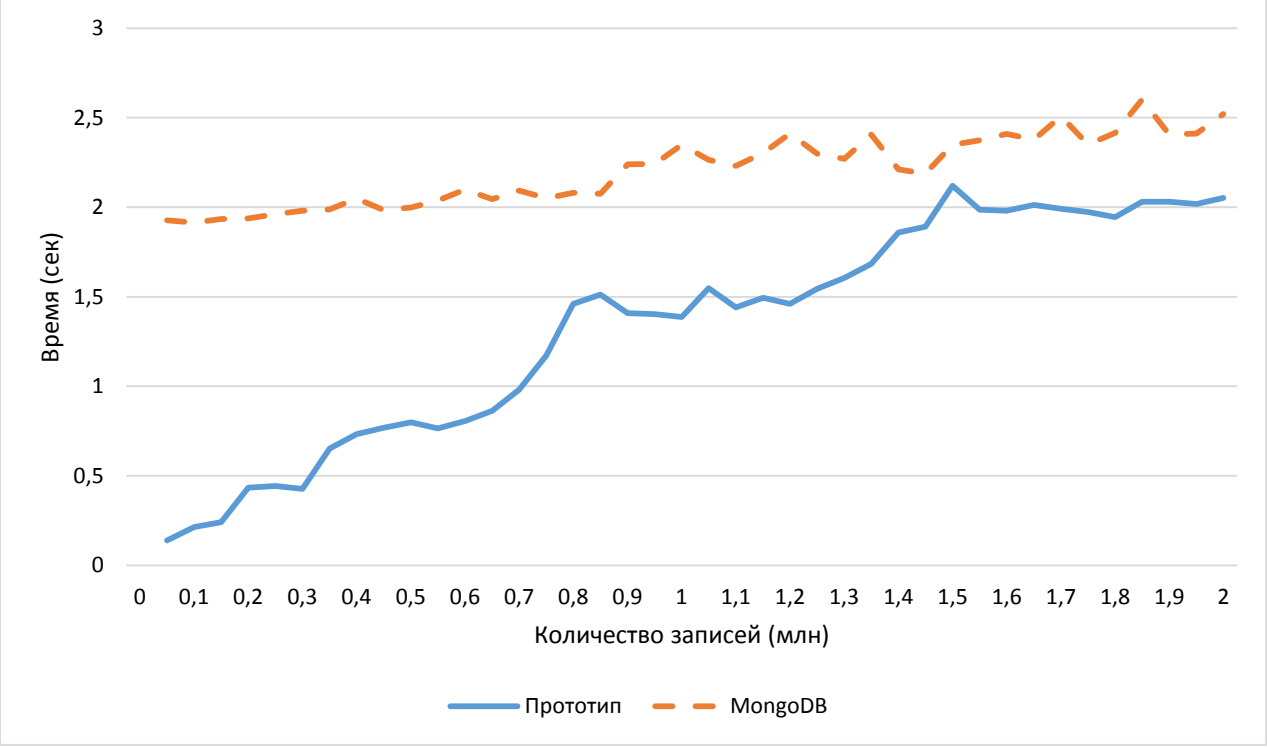


Рис. 4.7. Поиск заданной точки

Запросы к данным с совпадением отдельных координат. На рис. 4.8 и 4.9 изображены графики времени обработки операций поиска по одной и по двум заданным координатам соответственно. Заметим, что запросы поиска по одной координате построенный прототип обрабатывает медленнее чем MongoDB, однако при увеличении количества ключей поиска увеличивается также и его быстродействие. Это вызвано спецификой структуры сеточных файлов: при увеличении количества ключей поиска уменьшается количество сегментов, находящихся на пересечении соответствующих значениям заданных ключей поиска полос, и требуется меньшее количество дисковых операций для чтения данных.

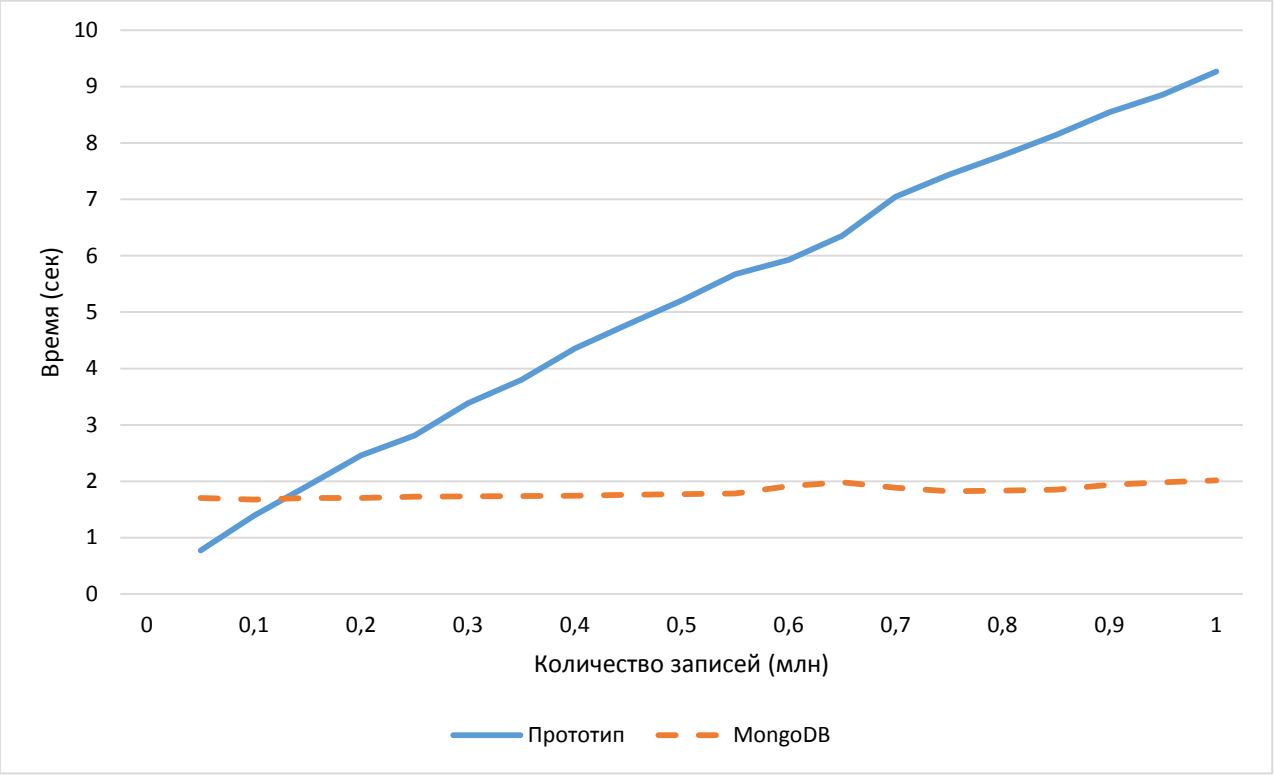


Рис. 4.8. Поиск по одной координате

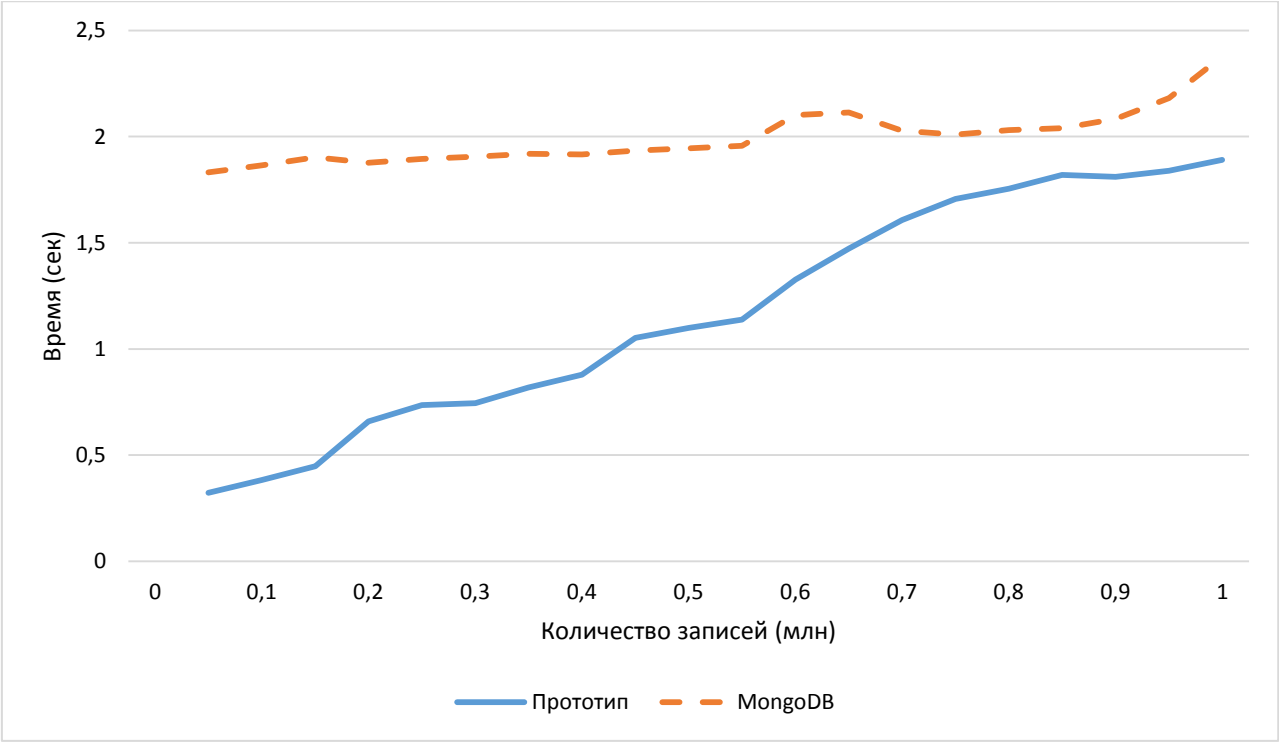


Рис. 4.9. Поиск по двум координатам

Запросы в диапазонах значений. Запрос в диапазоне значений определяет прямоугольную область сетки, и ответом на него является множество точек, принадлежащих сегментам, которые покрывают эту область. На рис. 4.10 и рис. 4.11 приведены графики времены обработки таких запросов в случаях, когда заданные диапазоны имеют среднюю длину 1000 и 1000000 соответственно. Можно видеть, что при увеличении размеров рассматриваемых диапазонов быстродействие построенного прототипа хранилища относительно MongoDB значительно возрастает.

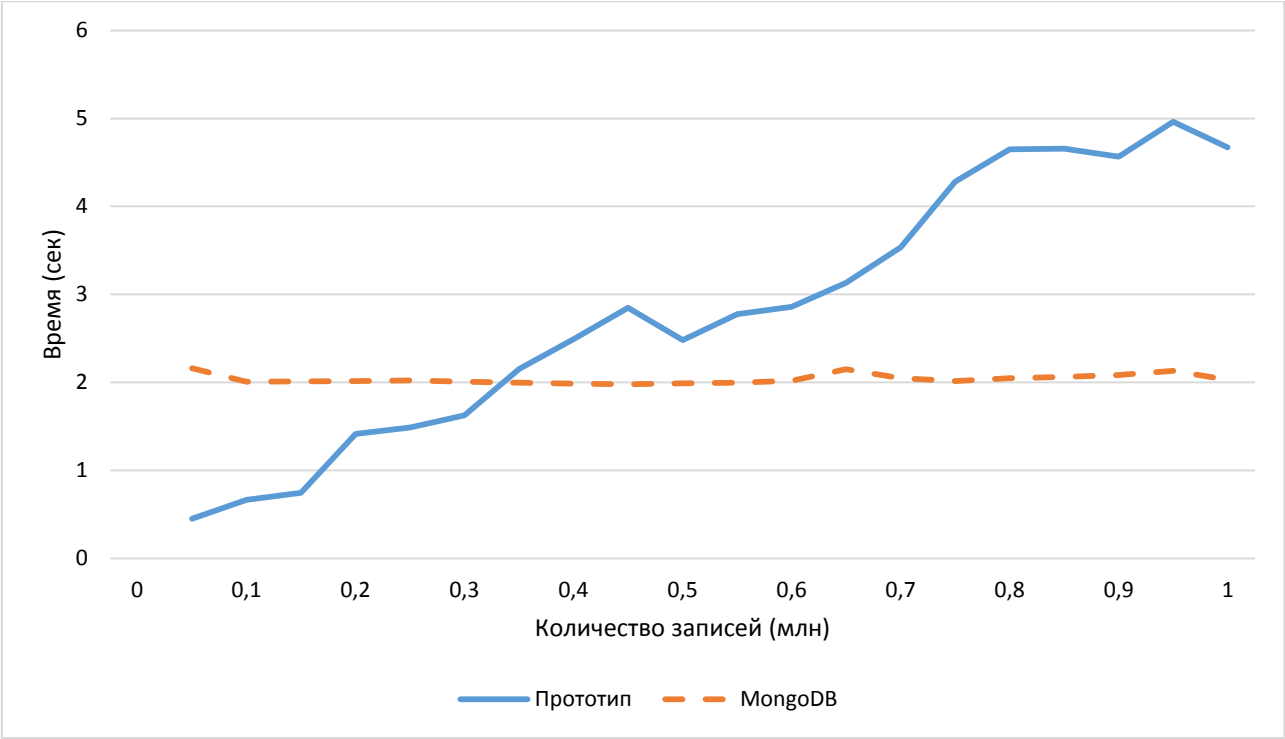


Рис. 4.10. Поиск в диапазонах средней длины 10^3

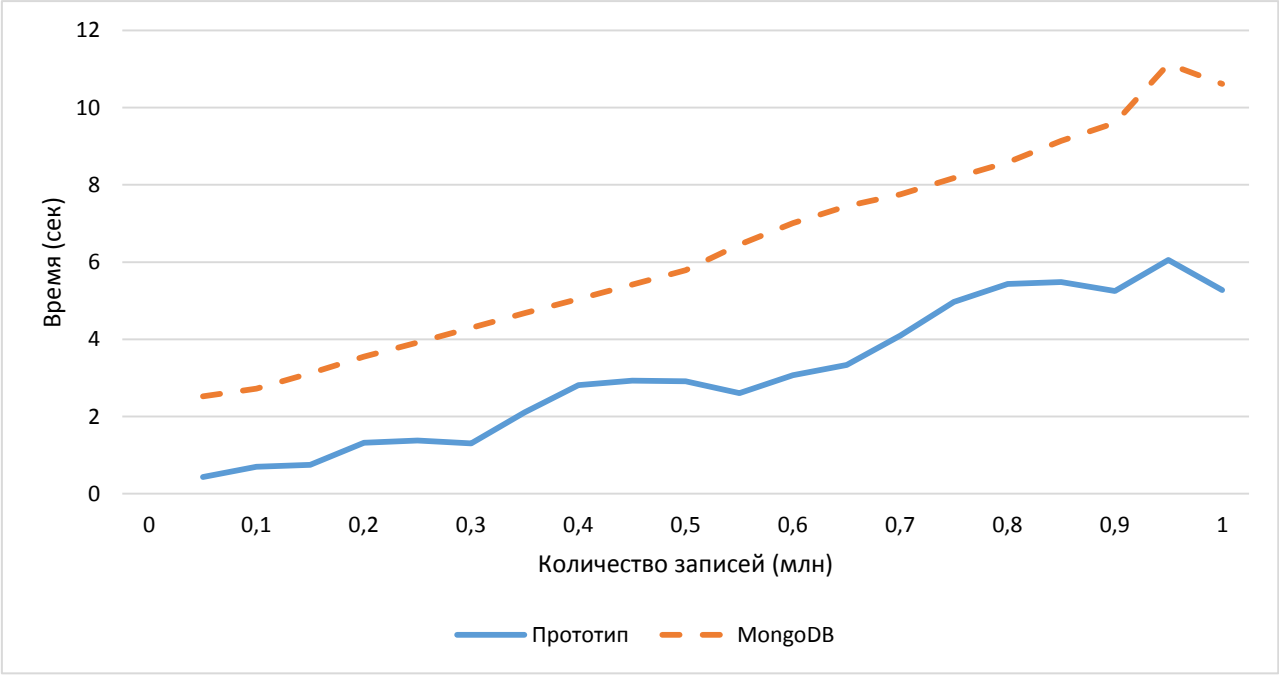


Рис. 4.11. Поиск в диапазонах средней длины 10^6

Поиск ближайших соседних объектов. Данный эксперимент был осуществлен с использованием точек двумерного евклидового пространства, чтобы сделать возможным использование *геопространственных индексов* (geospatial index) MongoDB. На рис. 4.12 можно видеть, что построенный прототип имеет значительное преимущество при выполнении запросов этого типа. Такое преимущество достигается благодаря возможности при необходимости быстро получать доступ к сегментам, граничащим с тем, в котором находится заданная точка, с помощью хранимых в самом сегменте указателей.

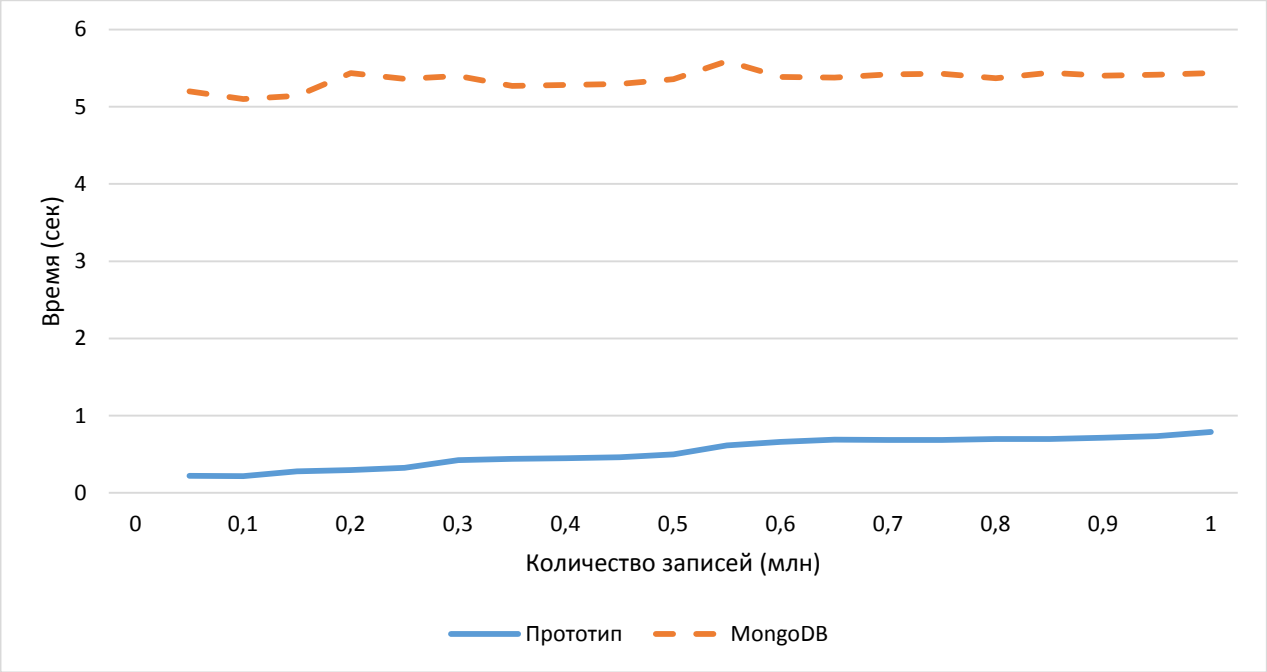


Рис. 4.12. Поиск ближайших соседних объектов

Таким образом, на основе проведенных экспериментов, показана эффективность построенного прототипа хранилища по сравнению с MongoDB в случаях поиска заданной точки, поиска в широких диапазонах значений, поиска ближайших соседних значений, а также более эффективное использование памяти.

4.3 Выводы по главе

В главе 4 разработан и реализован в среде языка C++ прототип хранилища данных на основе предложенной в главе 3 динамической структуры индекса для многомерных данных.

- Описана архитектура разработанного программного обеспечения, UML диаграмма классов и описание их функциональности;
- На основе проведенных экспериментов показана эффективность построенного прототипа хранилища по сравнению с MongoDB в случаях поиска заданной точки, поиска в широких диапазонах значений, поиска ближайших соседних значений, а также более эффективное использование памяти.

ЗАКЛЮЧЕНИЕ

Диссертационная работа посвящена исследованию задач интеграции данных, разработке подхода к интеграции данных на основе принципа коммутативных отображений моделей данных, включающего в себя разработку расширяемой канонической модели и метода верификации корректности отображений моделей данных, а также алгоритмов поддержки виртуальной и материализованной интеграции. В работе получены следующие результаты:

1. Разработан подход к интеграции данных на основе принципа коммутативных отображений моделей данных;
2. Предложена расширяемая XML-каноническая модель данных;
3. Построены AMN-машины для канонической и реляционной моделей данных и их обратимого отображения и доказана его корректность;
4. Разработан подход к построению хранилища данных, основанный на динамической структуре индекса для многомерных данных, предложены эффективные алгоритмы для ее поддержки и приведены оценки сложности предложенных алгоритмов;
5. Разработан и реализован прототип хранилища данных на основе предложенной динамической структуры индекса.

ЛИТЕРАТУРА

[1] A. Y. Halevy, "Answering queries using views: A survey," *VLDB Journal*, vol. 10, no. 4, pp. 270-294, 2001.

[2] M. Lenzerini, "Data integration: A theoretical perspective," in *Proceedings of Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Madison, Wisconsin, USA, 2002, pp. 233-246.

[3] M. Friedman, A. Y. Levy, and T. D. Millstein, "Navigational plans for data integration," in *Proceedings of Sixteenth National Conference on Artificial Intelligence and Eleventh Conference*, Florida, USA, 1999, pp. 63–73.

[4] A. Cali, "Reasoning in data integration systems: why lav and gav are siblings," in *Proceedings of ISMIS'03*, Maebashi City, Japan, 2003, pp. 562–571.

[5] http://synthesis.ipi.ac.ru.

[6] С. Ступников, Н. Скворцов, В. Буздко, В. Захаров, и Л. Калиниченко, "Методы унификации нетрадиционных моделей данных," *Системы высокой доступности, Радиотехника*, сс. 18-39, 2014.

[7] Л.А. Калиниченко, С.А. Ступников, и Н.А. Земцов, "Методы синтеза канонических моделей, предназначенных для достижения семантической интероперабельности неоднородных источников информации," Москва, 2005.

[8] P. Atzeni, L. Bellomarini, and F. Bugiotti, "Exlengine: Executable schema mappings for statistical data processing," in *Proceedings of EDBT'13*, New York, NY, USA, 2013, pp. 672-682.

[9] L. Bellomarini, P. Atzeni, and L. Cabibbo, "Data integration with many heterogeneous sources and dynamic target schemas (extended abstract)," in *Proceedings of AMW'15*, Lima, Peru, 2015, pp. 148-155.

[10] H Garcia-Molina, J Ullman, and J Widom, *Database Systems: The Complete Book.*: Prentice Hall, 2009.

[11] S. Sharma, U. S. Tim, J. Wong, S. Gadia, and S. Sharma, "A Brief Review on Leading Big Data Models," in *Data Science Journal,* December, 2014.

[12] J.-R. Abrial, *The B-Book - Assigning programs to meaning.*: Cambridge University Press, 1996.

[13] М. Г. Манукян и Г. Р. Геворгян, "Об одном подходе к задаче интеграции

информации," *Сборник статей VI Годичной Научной Конференции РАУ*, 2011, сс. 85-93.

[14] M. G. Manukyan and G. R. Gevorgyan, "An XML Mediator Based on the AMN Formalism," *Russian-Armenian (Slavonic) University Bulletin*, vol. 1, pp. 3-18, 2012.

[15] M. G. Manukyan and G. R. Gevorgyan, "An Approach to Information Integration Based on the AMN Formalism," in *Proceedings of First Workshop on Programming the Semantic Web, ISWC'12*, Boston, 2012. https://web.archive.org/web/20121225010018/http://www.inf.puc-rio.br/~psw12/papers.html

[16] G. R. Gevorgyan, "An Approach to Support Grid Files," in *Proceedings of X Annual Scientific Conference of RAU*, 2015.

[17] G. R. Gevorgyan and M. G. Manukyan, "Effective Algorithms to Support Grid Files," *Russian-Armenian (Slavonic) University Bulletin*, vol. 2, pp. 22-38, 2015.

[18] G. R. Gevorgyan, "An Effective Dynamic Structure for Grid file Organization," *Russian-Armenian (Slavonic) University Bulletin*, vol. 1, pp. 5-17, 2016.

[19] A. Y. Halevy, A. Rajaraman, and J. J. Ordille, "Data integration: The teenage years," in *Proceedings of 32nd International Conference on VLDB*, Seoul, Korea, 2006, pp. 9-16.

[20] J. D. Ullman, "Information integration using logical views," in *Proceedings of Database Theory - ICDT '97, 6th International Conference*, Delphi, Greece, 1997, pp. 19-40.

[21] http://www.oracle.com/technetwork/middleware/data-integrator/overview/index.html.

[22] http://www.oracle.com/technetwork/middleware/goldengate/overview/index.html.

[23] http://www-01.ibm.com/software/data/db2/.

[24] E. Codd, "A relational model for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377-387, 1970.

[25] E. Codd, "Relational database: A practical foundation for productivity," *Communications of the ACM*, vol. 25, no. 2, pp. 109-117, 1982.

[26] D. Tsichritzis and F. Lochovski., *Data Models*. Englewood Cliffs, New Jersey: Prentice-Hall, 1982.

[27] The SciDB Development Team, "Overview of SciDB," in *Proceedings of SIGMOD'10*, 2010.

[28] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The Architecture of SciDB," in *Proceedings of SSDBM'11*, 2011, pp. 1-16.

[29] http://paradigm4.com.

[30] P. Cudre-Mauroux et al., "A demonstration of scidb: a science-oriented dbms," in *Proceedings of VLDB Endowment*, vol. 2, 2009, pp. 1534-1537.

[31] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of OSDI'04*, 2004, pp. 137-150.

[32] P. Atzeni, F. Bugiotti, and L. Rossi, "Uniform access to NoSQL systems," *Information Systems*, vol. 43, pp. 117-133, 2014.

[33] G. Harrison. (2010) 10 things you should know about NoSQL databases. http://www.techrepublic.com/blog/10-things/10-things-you-should-know-about-nosql-databases/

[34] C. Mohan, "History repeats itself: sensible and NonsenSQL aspects of the NoSQL hoopla," in *Proceedings of EDBT'13*, 2013, pp. 11-16.

[35] M. Stonebraker and R. Cattell, "10 rules for scalable performance in 'simple operation' datastores," *Communications of the ACM*, vol. 54, no. 6, pp. 72-80, 2011.

[36] M. Stonebraker et al., "The end of an architectural era: (it's time for a complete rewrite)," in *Proceedings of VLDB'07*, 2007, pp. 1150-1160.

[37] http://www.couchbase.com.

[38] J. C. Anderson, N. Slater, and J. Lehnardt, "CouchDB: The Definitive Guide," 2009.

[39] F. Chang et al., "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 4:1-4:26, 2008.

[40] http://cassandra.apache.com.

[41] http://neo4j.com.

[42] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. USA: O'Reilly, 2015.

[43] https://www.mongodb.org.

[44] J. Nievergelt and H. Hinterberger, "The Grid File: An Adaptable, Symmetric, Multikey File Structure," *ACM Transactions on Database Systems*, vol. 9, no. 1, pp. 38-71, 1984.

[45] A. N. Papadopoulos, Y. Manolopoulos, Y. Theodoridis, and V. Tsoras, "Grid File (and

family)," *Encyclopedia of Database Systems*, pp. 1279-1282, 2009.

[46] P. Atzeni, F. Bugiotti, and L. Rossi, "Uniform access to non-relational database systems: The SOS platform," *Advanced Information Systems Engineering*, pp. 160-174, 2012.

[47] P. Atzeni, L. Bellomarini, F. Bugiotti, F. Celli, and G. Gianforme, "A runtime approach to model-generic translation of schema and data," *Information Systems*, pp. 269-287, 2012.

[48] P. Atzeni, L. Bellomarini, F. Bugiotti, and G. Gianforme, "A runtime approach to model-independent schema and data translation," in *Proceedings of EDBT'09*, 2009, pp. 275-286.

[49] P. Atzeni, P. Cappelari, R. Torlone, P. Bernstein, and G. Gianforme, "Model-independent schema translation," *VLBD Journal*, vol. 17, no. 6, pp. 1347-1370, 2008.

[50] C. K. Baru et al., "Xml-based information mediation with mix," in *Proceedings of SIGMOD'99*, Philadelphia, 1999.

[51] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld, "An adaptive query execution engine for data integration," in *Proceedings of ACM SIGMOD*, 1999, pp. 299-310.

[52] Z. Ives, A. Halevy, and D. Weld, "An xml query engine for network-bound data," *VLDB Journal, Special issue on XML Query Processing*, 2003.

[53] I. Manolescu, D. Florescu, and D. Kossmann, "Answering xml queries on heterogenous data sources," in *Proceedings of VLDB*, 2001, pp. 241-250.

[54] J. Naughton et al., "The Niagara Internet query system," *IEEE Data Engineering Bulletin*, June 2001.

[55] C. Yu and L. Popa, "Constraint-based xml query rewriting for data integration," in *Proceedings of SIGMOD'04*, 2004, pp. 371-382.

[56] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanascu, "XQuery: A query language for XML," 2001.

[57] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "A query language for XML," in *Proceedings of World Wide Web 8 Conference*, 1999, pp. 1155-1169.

[58] S. Chawathe et al., "The TSIMMIS project: Integration of heterogenous information sources," in *Proceedings of IPSJ'94*, 1994.

[59] P. A. Bernstein and S. Melnik, "Model management 2.0: manipulating richer mappings," in *Proceedings of SIGMOD'07*, 2007, pp. 1-12.

[60] J. F. Terwilliger, S. Melnik, and P. A. Bernstein, "Language-integrated querying of XML data in SQL server," in *Proceedings of VLDB'08*, 2008, pp. 1396-1399.

[61] P. Mork, P. Bernstein, and S. Melnik, "A schema translator that produces object-to-relational views," MSR-TR-2007-36, 2007.

[62] R. Cattell, "Scalable SQL and NoSQL data stores," in *Proceedings of SIGMOD Record*, vol. 39, 2010, pp. 12-27.

[63] M. Stonebraker, "Stonebraker on NoSQL and enterprises," *Communications of the ACM*, vol. 54, pp. 10-11, 2011.

[64] L. A. Kalinichenko, "Data model transformation method based on axiomatic data model extension," in *Proceedings of 4th International Conference on VLDB*, 1978, pp. 549-555.

[65] L. A. Kalinichenko, *Methods and Tools for Integration of Heterogeneous Databases (in Russian).*: Science, 1983.

[66] L. A. Kalinichenko, "Methods and tools for equivalent data model mapping construction," in *Proceedings of Advances in Database Technology-EDBT'90*, 1990, pp. 92-119.

[67] L. A. Kalinichenko, D. O. Briukhov, D. Martynov, N. A. Skvortsov, and S. A. Stupnikov, "Mediation framework for enterprise information system infrastructures: Application-driven approach," in *Proceedings of Ninth International Conference on Enterprise, ICEIS*, Funchal, Portugal, 2007, pp. 246–251.

[68] L. A. Kalinichenko and S. A. Stupnikov, "Owl as yet another data model to be integrated," in *Proceedings of 15th East European Conference, ADBIS*, Vienna, Austria, 2011, pp. 178-189.

[69] Andreas Zelend, "Formal Product Families for Abstract Machines," Institute of Informatics, University of Augsburg, Augsburg, 2011.

[70] J.-R. Abrial, "B-Technology: Technical overview," 1992.

[71] J.-R. Abrial, D. Cansell, and G. Laffitte, "Higher-Order Mathematics in B," *Lecture Notes in Computer Science*, vol. 2272, pp. 370–393, 2002.

[72] D. Cansell and D. Mery, "Foundations of the B-method," *Computing and Informatics*, vol. 22, pp. 1-31, 2003.

[73] L. A. Kalinichenko, "Method for data models integration in the common paradigm," in *Proceedings of First East European Conference, ADBIS*, St.-Petersburg, Russia, 1997,

pp. 275-284.

[74] L. A. Kalinichenko and S. A. Stupnikov, "Constructing of mappings of heterogeneous information models into the canonical models of integrated information systems," in *Proceedings of Advances in Databases and Information Systems: Proc. of the 12th East-European Conference*, Pori, Finland, 2008, pp. 106-122.

[75] M. G. Manukyan, "Extensible Data Model," in *Proceedings of Advances in Databases and Information Systems*, 2008, pp. 42-57.

[76] M. G. Manukyan, "Canonical Model: Construction Principles," *iiWAS2014*, pp. 320-329, 2014.

[77] Hindley and J. P. Seldin, *Introduction to Combinators and λ-Calculus.*: Cambridge University Press, 1986.

[78] M. G. Manukyan, "Element Algebra," in *Proceedings of Advances in Databases and Information Systems, Associated Workshops and Doctoral Consortium of the 13th East European Conference, ADBIS*, Riga, Latvia, 2009, pp. 113-120.

[79] M. G. Manukyan, "Element algebra (extended version)," *International Journal of Information Technology and Database Systems*, vol. 1, no. 1, pp. 43-56, 2010.

[80] M. Drawar, "OpenMath: An Overview," *ACM SIGSAM Bulletin*, vol. 34, no. 2, 2000.

[81] http://www.openmath.org/.

[82] http://www.atelierb.eu/en/.

[83] C. Luo, W. C. Hou, C. F. Wang, H. Want, and X. Yu, "Grid File for Efficient Data Cube Storage," *Computers and their Applications*, pp. 424-429, 2006.

[84] K.-Y. Whang and R. Krishnamurthy, "The Multilevel Grid File - A Dynamic Hierarchical Multidimensional File Structure," in *Proceedings of DASFAA Conference*, 1991, pp. 449-459.

[85] M. Regnier, "Analysis of Grid File Algorithms," *BIT*, vol. 25, no. 2, pp. 335-358, 1985.

[86] E. J. Otoo, "A Mapping Function for the Directory of a Multidimensional Extendible Hashing," in *Proceedings of 10th International Conference on VLDB*, Singapore, 1984, pp. 493-506.

[87] E. J. Otoo, "A multidimensional digital hashing scheme for files with composite keys," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1985, pp. 214-229.

[88] N. N. Karayannidis, *Storage Structures, Query Processing, and Implementation of On-Line Analytical Processing Systems (Ph.D. Thesis)*. Athens, 2003.

[89] R. Kaas, E. Zijlstra, and P. Van Emde Boas, "Design and implementation of an efficient priority queue," *Mathematical Systems Theory*, 1976.

[90] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms (third edition).*: MIT Press, 2009.

[91] B. Ding and A. C. Konig, "Fast Set Intersection in Memory," in *Proceedings of 37th International Conference on VLDB*, Seattle, 2011, pp. 255-266.

[92] https://github.com/alveko/keithschwarz_cpp/blob/master/van-emde-boas-tree.