И. И. Труб, канд. техн. наук, доцент, вед. инженер-программист, ООО «Исследовательский центр Samsung», Москва, itrub@yandex.ru

Имитационное моделирование иерархических bitmap-индексов

В статье рассмотрено построение имитационной модели для иерархических bitmap-индексов на языке C. Индексы строятся по свойству, являющемуся значением времени занесения записи в базу данных. Модель позволяет проектировщику выбрать наиболее эффективную иерархию индексов по критерию минимизации логических операций при выполнении поисковых запросов. Отдельное внимание уделено верификации модели путем сравнения с частными случаями известных аналитических решений.

Ключевые слова: иерархические bitmap-индексы, дизъюнкция, исключающее ИЛИ, случайный поток событий, плотность распределения вероятностей, имитационное моделирование, доверительный интервал.

Введение

Bitmap-индексы являются широко распространенным встроенным инструментом СУБД для повышения производительности обработки поисковых запросов. Они используются в таких известных СУБД, как Oracle и Cache [6, 12, 20]. Напомним, что использование bitmap-индексов для некоторого столбца/свойства заключается в создании отдельных битовых строк для каждого возможного значения свойства, где каждому биту соответствует строка/объект (далее для определенности будем пользоваться термином *запись*) таблицы базы данных с этим значением. Если бит равен 1, то это означает, что запись, соответствующая позиции бита, содержит индексируемое значение для данного свойства. Описание реализации и использования битовых индексов в Cache можно найти в [7], в Oracle — в [15]. Типичным примером является генерация отчетов по значению свойства, являющегося временем занесения записи в базу данных. Например, в CRM-системе (Customer Relationship Management) имеется некоторая сущность, скажем, «Входящие звонки», экземпляры которой хранятся в отдельной структуре (для Cache таковой является глобал, но можно представлять ее и таблицей — общая постановка задачи никак не будет привязана к особенностям конкретной СУБД). Одним из атрибутов сущности является время поступления звонка с точностью до секунды, причем упорядоченности записей таблицы по этому атрибуту разработчик ожидать не вправе, т. к. информация о звонке заносится в БД не в момент поступления звонка, а позже. Интенсивность поступления звонков в масштабах всей компании, эксплуатирующей приложение, достаточна высока, т. е. объем таблицы велик. Типовым сервисом приложения, востребованным заказчиком, является генерация отчетов по входящим звонкам, причем основным фильтром при задании параметров отчета служит временной диапазон, например, «Вывести информацию обо всех звонках, поступивших с такого-то времени по такое». Для этого примера использование bitmap-индексов означает, что для каждого уникального значения времени, для которого зафиксирован хотя бы один звонок, создается битовая строка, возможно (в случае большого

количества записей), состоящая из нескольких подстрок (chunks). При запросе на отчет берутся все имеющиеся индексы для значений, входящих в заданный пользователем диапазон, и путем их дизъюнкции вычисляется результирующая битовая строка, единицы которой соответствуют ID записей, удовлетворяющих запросу и выводимых в итоговый отчет.

Однако практика показывает, что даже применение bitmap-индексов для промышленных объемов данных не дает приемлемой для потребителя производительности — отчеты генерируются слишком медленно. Ключевым параметром является в данном случае количество операций дизъюнкции между битовыми строками, т. е. количество индексов, подпадающих под условия фильтра запроса. При большой величине временнóго диапазона в фильтре и высокой интенсивности занесения в БД новых записей оно оказывается очень велико, что и приводит к ощутимой задержке. В этой связи актуальны расширения и улучшения технологии, одной из которых является концепция иерархических bitmap-индексов. Оценить эффективность применения этой концепции можно различными способами, например, с помощью имитационного моделирования. Построению и анализу результатов такой имитационной модели и посвящена настоящая статья. Она построена следующим образом. В первой части описаны иерархические bitmap-индексы применительно к временнóму свойству, дан краткий обзор соответствующей литературы, введены используемые обозначения и поставлена задача построения имитационной модели. Во второй части представлена сама модель и алгоритмы, легшие в ее основу. В третьей части описаны особенности анализа выходных данных модели. В четвертой части приведены примеры реализации модели для различных вариантов входных данных. В заключении указаны направления дальнейшего расширения и развития модели.

Иерархические bitmap-индексы

Предположим, что занесение новых записей в конкретную таблицу базы данных образует поток событий в непрерывном времени, описываемый функцией распределения случайной величины X_1 интервала между двумя последовательными событиями $F_1(t)$, а длина временнóго интервала, задаваемого в запросе на выборку записей, является случайной величиной X_2 с распределением $F_2(t)$. Шкала времени разбита на равные интервалы единичной длины. Начало интервала для X_2 всегда совмещено с началом одного из единичных интервалов. Назовем полуоткрытый единичный интервал $[i;\ i+1)$ *помеченным*, если он содержит хотя бы одно событие из потока X_1. Дискретную случайную величину Y определим как количество помеченных интервалов, полностью накрываемых на временнóй шкале случайным интервалом X_2. Тогда Y как раз и является количеством индексов (дизъюнкций), используемых для вычисления результата запроса. Чем меньше эта величина, тем меньше времени занимает обработка запроса. Аналитическая модель вычисления распределения для Y была построена в [8] и [9]. При этом в [8] в предположении экспоненциальности $F_1(t)$ было получено точное решение, а в [9] для произвольной $F_1(t)$ — численное (вид $F_2(t)$ не оказывает принципиального влияния на сложность аналитической модели).

Идея иерархических bitmap-индексов применительно ко времени заключается в следующем: количество индексов, необходимых для получения результата запроса, можно уменьшить, если наряду с *мелким* индексом, соответствующим основной единице времени (секунде), СУБД будет поддерживать еще некоторое количество *крупных* индексов, соответствующих интервалам времени, кратным основному. Например, если наряду с посекундными индексами мы располагаем также и поминутными, то для обслуживания запроса длиной в минуту (если его левая гра-

ница совпадает с границей минуты) нам понадобится максимум одна индексная битовая строка, а не 60. Идея проста и понятна, но ее использование требует точных определений и обоснованной методики расчета показателей эффективности. В частности, необходимо с той или иной точностью дать ответ на следующие вопросы:

- как вычислить распределение случайной величины Y для заданной иерархии крупных индексов;
- как выбрать наиболее эффективную иерархию индексов и какие критерии эффективности использовать.

Понятие иерархических bitmap-индексов (hierarchical bitmap indexes) было впервые введено в [21] для индексирования свойств, являющихся множествами значений некоторых простых типов (set-valued attribute). Тот смысл, в котором термин используется в настоящей работе, известен в англоязычной литературе также как *binning* — объединение отдельных значений данных в группы по некоторому правилу и создание укрупненных индексов по этим группам. Обзор результатов по binning с подробным объяснением основных концепций на примерах содержится в [23]. Отмечено, что данная технология хорошо подходит для больших массивов результатов различных научных и технических измерений, таких как давление, температура и пр. Близко к рассматриваемым задачам примыкает сравнительно недавняя работа американских специалистов [22]. В ней рассматривается задача выбора между OR- и XOR-операциями, которые названы *inclusive query plan* и *exclusive query plan*, соответственно; сформулирована проблема оптимального выбора плана запроса (*cut selection problem*), предложены некоторые эвристические алгоритмы ее решения, применение которых проиллюстрировано на синтетических наборах данных. Но в этой статье рассмотрена оптимизация запроса на конкретной иерархии индексов, а не оптимальный выбор самой иерархии. В [10], являющейся первой работой подобного рода, задача оценки эффективности использования иерархических bitmap-индексов решена с помощью формально обоснованной математической модели методами теории вероятностей. В этой работе, а также в [14], описанная выше постановка задачи была продолжена на иерархию индексов и построена модель, в которой распределение и среднее для случайной величины Y доводятся до вычисленных по выведенным формулам значений. Следуя [10], приведем эту постановку, необходимую для дальнейшего понимания работы.

Определим *иерархию индексов* как упорядоченное множество $K = \{k_i\}$, $i = 1,\ldots, N$, где каждый элемент множества является натуральным числом больше единицы. Смысл этого понятия заключается в следующем: наряду с мелким индексом поддерживаются N крупных индексов, кратность первого по отношению к мелкому равна k_1, кратность второго по отношению к первому равна k_2 и т. д. N назовем *размером иерархии*. Например, $K = \{60, 60\}$ соответствует совокупности трех индексов размерами одна секунда, одна минута и один час.

Введем также множество $L = \{l_i\}$, $i = 0,\ldots, N$, где $l_0 = 1$, $l_i = \prod_{j=1}^{i} k_i$, $i > 0$. Из определения очевидно, что i-й элемент этого множества есть размер i-го крупного индекса относительно основной единицы измерения — мелкого индекса. В [10] и [14] рассматривались только такие интервалы времени, левая граница которых совпадает с границей самого крупного индекса, имеющего в иерархии K порядковый номер, равный размеру иерархии N. В примере {60, 60} это начало любого часа. Это предположение было названо *элементарным граничным условием* (*ЭГУ*), и его выполнение позволяет любой интервал времени длины T представить единственным образом в виде упорядоченного множества $B = \{b_i\}$, $i = 0,1,\ldots, N$, где i-й элемент множества есть количество единиц i-го индекса иерархии, а $T = \sum_{j=0}^{N} b_j l_j$. Такое представление

позволило построить вероятностную модель иерархии индексов и рассчитать ее.

Итак, любому запросу на выборку данных по времени при условии выполнения ЭГУ соответствует некоторое множество B разложения длины запроса по иерархии индексов. Очевидно, что значение b_N может быть произвольным неотрицательным целым числом, а для всех $0 \le i \le N - 1$ должно выполняться соотношение $0 \le b_i \le k_{i+1}$, т. к. количество индексов i-го уровня не может быть больше соответствующей ему кратности следующего уровня. Однако такая интерпретация модели является не вполне экономичной. Поясним это примером. Предположим, что требуется построить выборку, где начало интервала совпадает с началом минуты, а длина интервала равна 45 сек. Если вычислять результат запроса исключительно с помощью дизъюнкций (*OR-operation* или *inclusive query plan*), потребуется дизъюнкция 45 индексных битовых строк — от нулевой до 44-й сек. Но их количество можно сократить до 16, если последовательно осуществить операцию «исключающее ИЛИ» (*XOR-operation* или *exclusive query plan*) полного минутного индекса с индексами от 45-й до 59-й сек. Иными словами, вместо «складывания» событий, произошедших от 0 до 44-й секунды, из всех событий, произошедших в течение минуты, следует «вычесть» те события, которые в этот интервал не попали. Формализация указанного соображения приводит к следующей связи между множествами B и K:

$$0 \le b_i \le R_i,\ i = 0, \ldots, N-1,$$

$$R_i = \begin{cases} \dfrac{k_{i+1}}{2},\ k_{i+1} - \text{четное} \\ \dfrac{k_{i+1}+1}{2},\ k_{i+1} - \text{нечетное} \end{cases}.$$

Интуитивно очевиден тот факт, что, с одной стороны, чем больше размер иерархии N, тем меньше будет значение $Y_{\text{ср.}}$, т. е. тем меньше в среднем потребуется использовать битовых индексных строк для обслуживания запроса. Но, с другой стороны, поддержание лишнего крупного индекса требует определенных накладных расходов от СУБД, т. к. этот индекс нужно обновлять при каждой операции добавления и удаления записей, кроме того, требуется место, чтобы его хранить. Поэтому вполне естественно, что на размер иерархии должны накладываться некоторые ограничения. Это позволило поставить в [10] содержательные проблемы условной оптимизации по выбору наилучшей в некотором смысле иерархии K.

Но для решения любой задачи оптимизации необходим движок — метод вычисления значения минимизируемой функции в заданной точке. В нашей задаче это вычисление значения $Y_{\text{ср.}}$ для заданной иерархии K, что представляет собой самостоятельную и весьма непростую задачу. Как уже было сказано, эта задача была решена аналитически в [10] при наличии двух упрощающих предположений: все запросы удовлетворяют ЭГУ и распределение $F_1(t)$ является экспоненциальным. Однако на практике ожидание выполнения этих ограничений не вполне оправдано. Записи могут искаться в интервале, не начинающемся с границы часа, а входной поток записей в базу данных не будет пуассоновским. Ввиду того, что отбрасывание любого из них сразу же разрушает построенную в [10] модель, решение задачи в общем виде возможно только средствами имитационного моделирования, т. к. имитационная модель позволяет реализовать любую внутреннюю логику системы. Таким образом, точка, в которой вычисляется значение минимизируемой функции — это множество K, а в качестве процедуры вычисления выступает прогон имитационной модели.

Имитационная модель

Рассмотренная задача имеет особенности, отличающие ее от традиционных задач на построение имитационной модели. В первую очередь моделируемая система не является

системой массового обслуживания (СМО) в классическом смысле этого термина. Что имеется в виду? В СМО необходимо разыгрывать две последовательности случайных величин — интервалы между поступлениями заявок и длительности обслуживания заявок. И обе эти последовательности можно разыгрывать в динамике, т. е. в момент поступления в систему очередной заявки генерируется по очередному члену каждой последовательности. В нашем же случае логика несколько иная. Для моделирования запроса на выборку данных необходимо, чтобы значения временнóго атрибута были уже назначены всем записям базы данных. Т.е. сначала в полном объеме должна быть сформирована реализация распределения $F_1(t)$ (будем для краткости называть ее *поле*), и только после этого реализуется распределение $F_2(t)$. Очередное значение искомой случайной величины Y вычисляется для каждого элемента этой реализации, после чего реализуется следующий элемент. Иными словами, имея поле, иерархию индексов и интервал запроса, мы по определенному алгоритму, являющемуся собственно алгоритмическим движком моделирования, вычисляем, сколько индексов понадобится для обслуживания этого запроса. Указанная особенность сводит имитационный эксперимент к вычислению $Y_{\text{ср.}}$ как среднего значения последовательности $Y_{\text{ср.},1}$, $Y_{\text{ср.},2}$, …, где $Y_{\text{ср.},i}$ — среднее значение для реализации Y, вычисленной на одном и том же поле. Назовем эту последовательность *глобальной*, в то время как последовательность реализаций Y на одном и том же поле для каждого i — *локальной*.

Оригинальность моделируемой системы делает малопригодным использование каких-либо стандартных универсальных пакетов, таких как AnyLogic [2, 5], т. к. применение к данной задаче либо невозможно, либо требует самостоятельного написания большого количества кода на языке JAVA. Поэтому для создания адекватной модели был использован подход, близкий к тому, на котором основаны работы [11] и [13], — собственная реализация на универсальном языке программирования C/C++, позволяющая полностью учесть все алгоритмические нюансы задачи. По существу, процесс создания модели полностью, вплоть до языка программирования, соответствует классическому плану, изложенному в [1], а именно:

1) выбор объекта, системы процессов, которые необходимо смоделировать;
2) формулирование вопросов, на которые должна отвечать модель;
3) построение графа модели;
4) формулирование требований к внутренней логике;
5) кодирование графа модели на языке программирования C;
6) кодирование внутренней логики на C;
7) компиляция и сборка;
8) отладка модели.

Перейдем к рассмотрению архитектуры модели (рис. 1). На рисунке 1 отображены функции, из которых состоит программный код, и связи между ними, где наличие направленной связи обозначает факт вызова из одной функции другой функции. Опишем семантику этих функций.

- *cycleGlo* — цикл, на каждой итерации которого вычисляется очередное значение $Y_{\text{ср.},i}$;
- *batchingGlo*, *estimationGlo* — функции анализа выходных данных, используемые для определения момента прекращения моделирования, т. е. момента, когда доверительный интервал для $Y_{\text{ср.}}$ с требуемым уровнем достоверности уже получен. Подробнее они рассмотрены в третьей части работы;
- *spectralVarAnalysis* — функция, лежащая в основе анализа выходных данных;
- *leastSquares* — реализует интерполяцию методом наименьших квадратов;
- *getSingleDataGlo* — реализует вычисление одного значения $Y_{\text{ср.},i}$;
- *getGround* — вычисляет новое поле, т. е. последовательность целочисленных случайных величин, интервалы между которыми описываются функцией $F_1(t)$. Генерация поля

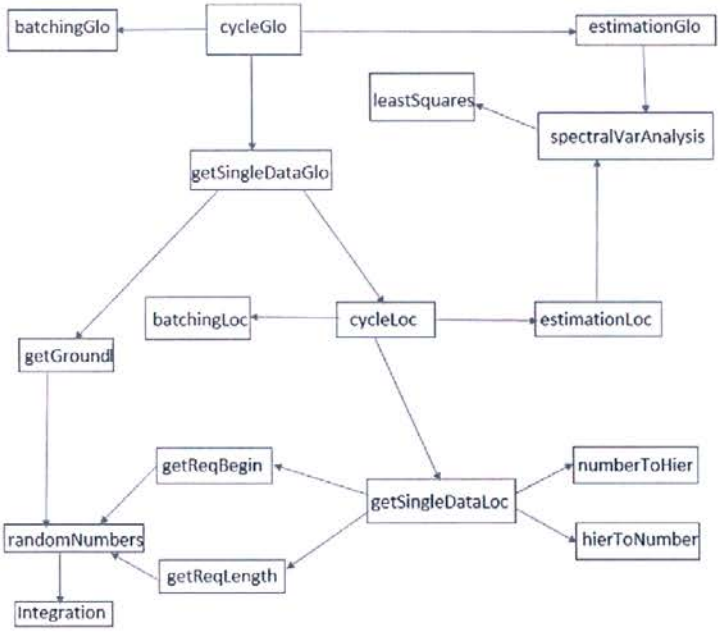

Рис. 1. Программная структура имитационной модели

Fig. 1. Structure of simulation software

продолжается до достижения значения 86 400 (количество секунд в сутках). В зависимости от условий и целей моделирования может быть выбрано и другое значение;

• *batchingLoc* и *estimationLoc* — анализ выходных данных в локальном цикле. Реализация та же, что и для соответствующих глобальных функций. Выделены в отдельные блоки, чтобы еще раз подчеркнуть двухуровневую структуру модели;

• *getSingleDataLoc* — собственно, движок моделирования. Генерирует очередной запрос на выборку данных и на основе моделируемой иерархии вычисляет количество индексов для его обслуживания;

• *numberToHier*, *hierToNumber* — вспомогательные функции, конвертирующие абсолютное значение времени в секундах в его иерархическое представление $B = \{b_i\}$, $i = 0,1,\ldots,N$ и наоборот;

• *getReqBegin* — генерирует левую границу интервала запроса при отсутствии выполнения ЭГУ. Алгоритм подробнее рассмотрен далее;

• *getReqLength* — генерирует длину запроса в соответствии с распределением $F_2(t)$;

• *randomNumbers* — генератор случайных чисел. Реализован с помощью метода обратной функции. Многочисленные примеры реализации на языке C приведены в [11];

• *Integration* — реализация численного интегрирования, необходимого генератору случайных чисел для вычисления некоторых функций распределения.

Опишем обобщенную математическую модель, используемую для генерации начальной точки интервала запроса при снятии ЭГУ. В примере иерархии секунда-минута-час это означает, что таковой может быть не только момент времени, кратный часу, а любой момент времени. Зададим два множества: $NB = \{NB_i\}$ и $PB = \{PB_i\}$, $i = 1,\ldots, Z$, удовлетворяющие условиям:

• NB_i нацело делится на NB_{i-1}, $i > 1$;

• $\sum_{i=1}^{Z} PB_i = 1$.

Смысл данной модели состоит в следующем: с вероятностью PB_i начало интер-

вала приходится на момент времени, который в абсолютном исчислении (в сек.) кратен NB_i, но не кратен NB_{i+1}. Например, частный случай выполнения ЭГУ будет выражен так: $Z = 1, NB = \{l_N\}, PB = \{1.0\}$. Заметим, что множество NB не имеет никакого отношения к иерархии индексов и может быть задано совершенно независимо от множества K. В листинге 1 представлена реализация этой модели с комментариями.

Разберем далее алгоритм для функции *getSingleDataLoc*. Последовательным вызовом функций *getReqLength* и *numberToHier* получаем два момента времени в их иерархическом представлении $B^{(1)} = \{b_i^{(1)}\}$, $B^{(2)} = \{b_i^{(2)}\}$, $i = 0,..,N$. Теперь, зная иерархию индексов и располагая ранее сгенерированным полем, необходимо подсчитать, сколько индексов попадет в этот интервал. Введем понятие *седловой точки* интервала как $h(B^{(1)}, B^{(2)}) = \max(b_i^{(2)} > b_i^{(1)})$, и с ее помощью запишем принципиальный алгоритм этого подсчета.

1. $i := 0$, $s := 0$.

2. Сопоставлением с полем подсчитать количество индексов i-го уровня в интервале

$$(0,0,...,0,b_i^{(1)},b_{i+1}^{(1)},...,b_h^{(1)},...,b_N^{(1)}),$$
$$(0,0,...,0,0,b_{i+1}^{(1)}+1,...,b_h^{(1)},...,b_N^{(1)}).$$

Нарастить значение s.

3. $b_{i+1}^{(1)} := b_{i+1}^{(1)} + 1$.

4. $i := i + 1$. Если $i < h$, перейти к шагу 2. Иначе — к шагу 5.

Листинг 1. Функция getReqBegin ()

Listing 1. Source code of getReqBegin () function

```
int getRequestBegin() {
 int k,r_num,n1,n2,m;
 float s;
//генерируем случайное целое число до тысячи, чтобы разыграть какому элементу
// множества NB будет соответствовать начало интервала
 r_num = rand() % 1000;
 s=0.0;
 for (k=0;k<Z;k++) {
 s=s+Pb[k]*1000;
 if (r_num < s) break;
 break;
 }
// вычисляем n1 - количество моментов времени, соответствующих разыгранному k
 if (k==Z-1) n1 = 86400/Nb[k];
 else n1=86400/Nb[k] - 86400/Nb[k+1]; //уменьшаем на число кратных следующему
элементу
//теперь из n1 значений нужно выбрать какое-то одно случайным образом
 m = (rand() % n1) * Nb[k];
 if (k != (Z-1)) {
//из m чисел, делящихся на Nb[k], n2 делятся также и на Nb[k+1], поэтому нужно
отсчитать вперед после m еще n2 чисел, которые делятся на Nb[k], но не делятся
на Nb[k+1]
 n2 = m/Nb[k+1]+1;
 for(i=0;i<n2;i++) {
 m=m+Nb[k];
 if (m % Nb[k+1] == 0) m=m+Nb[k];
 }
 }
 return(m);
}
```

5. Сопоставлением с полем подсчитать количество индексов h-го уровня в интервале

$$(0,0,...,0,b_h^{(1)},b_{h+1},...,b_N),$$
$$(0,0,...,0,b_h^{(2)},b_{h+1},...,b_N)$$

(напомним, что согласно определению седловой точки $b_i^{(1)} = b_i^{(2)}$, при $i > h$). Нарастить значение s.

6. $i := h-1$.

7. Сопоставлением с полем подсчитать количество индексов i-го уровня в интервале $(0,0,...,0,b_{i+1}^{(2)},b_{i+2}^{(2)},...,b_N^{(2)})$, $(0,0,...,b_i^{(2)},b_{i+1}^{(2)},b_{i+2}^{(2)},...,b_N^{(2)})$. Нарастить значение s.

8. Если $i = 0$, конец, s — результат. Иначе $i := i-1$ и перейти к шагу 7.

Отметим, что при выполнении шагов 2, 5 и 7 количество индексов подсчитывается с учетом OR- либо XOR-операции, в зависимости от того, каких будет меньше. Так, на шаге 2 моделируется OR-операция, если $b_i > k_{i+1}/2$, и XOR-операция — в противном случае. На шаге 7, соответственно, наоборот: XOR-операция, если $b_i > k_{i+1}/2$, и OR-операция — в противном случае. На шаге же 5, в случае $h < N$ правило таково: OR-операция в случае $2(b_h^{(2)} - b_h^{(1)}) \le k_{h+1} + 1$ и XOR-операция — в противном случае. Доказательство достаточно очевидно, предоставим читателю возможность провести его самостоятельно.

Проиллюстрируем особенности алгоритма некоторыми примерами. В иерархии (секунда-минута-час) пусть $B^{(1)} = (6,10,3)$, $B^{(2)} = (3,12,5)$. Здесь $N = 2$, $h = 2$. Согласно алгоритму на шаге 2 находим количество секундных индексов на интервале (6,10,3) — (0,11,3) XOR-операцией, т. к. $6 < 60/2 = 30$. Затем на шаге 2 находим количество минутных индексов на интервале (0,11,3) — (0,0,4) XOR-операцией. Затем на шаге 5 количество часовых индексов на интервале (0,0,4) — (0,0,5) (это будет 0 или 1, т. к. интервал — один час). Далее на шаге 7 находим количество минутных индексов на интервале (0,0,5) — (0,12,5) OR-операцией, затем на шаге 7 количество секундных индексов на интервале (0,12,5) — (3,12,5) OR-операцией. Таким образом, стартовав от левой границы интервала и действуя согласно алгоритму, мы финишировали на его правой границе.

Разберем далее пример изначального интервала (6,10,3) — (20,45,3). Здесь $h = 1$. На шаге 2 находим количество секундных индексов на интервале (6,10,3) — (0,11,3) XOR-операцией. Затем на шаге 5 находим количество минутных индексов на интервале (0,10,3) — (0,45,3) XOR-операцией,

Листинг 2. Функция numberToHier ()

Listing 2. Source code of numberToHier () function

```
void numberToHier(int a, int res[hierN+1]) {
//массив res соответствует множеству B, константный массив hierarchy - множе-
ству K
 int i, pr=1;
 for(i=0;i<hierN;i++) pr=pr*hierarchy[i]; //вычисляем размер самого крупного
индекса иерархии
 for(i=hierN;i>=0;i--) {
//последовательно раскладываем абсолютное значение времени по иерархии сверху
вниз
 res[i]=a/pr; //целая часть от деления
 if (i != 0) {
 a=a % pr; //остаток от деления
 pr=pr/hierarchy[i-1];
 }
 }
}
```

Листинг 3. Функция hierToNumber ()
Listing 3. Source code of hierToNumber () function

```
int hierToNumber(int res[hierN+1]) {
//массив res соответствует множеству B, константный массив hierarchy - множе-
ству K
 int pr=1,sum=0,i;
 for(i=0;i<=hierN;i++) {
 sum=sum+pr*res[i];
 if (i==hierN) break;
 pr=pr*hierarchy[i];
 }
 return(sum);
}
```

т. к. $2 \cdot (45-10) = 70 > k_2 + 1 = 61$. Затем на шаге 7 находим количество секундных индексов на интервале (0,45,3) — (20,45,3) OR-операцией.

В заключение второй части приведем коды вспомогательных функций и *numberToHier* (листинг 2) и *hierToNumber* (листинг 3), т. к. с одной стороны они невелики по объему, с другой помогут глубже понять детали рассматриваемой задачи.

Анализ выходных данных

Целью анализа выходных данных в рассматриваемой задаче является определение момента, когда моделирование следует прекратить, т. е. доверительный интервал для величины $Y_{ср.}$ с требуемым уровнем достоверности уже получен. Особенностью задачи является то, что этот анализ нужно проводить на двух уровнях — локальном (моделирование на одном поле) и глобальном (последовательность результатов моделирования на разных полях). Основы анализа выходных данных (*output analysis*) в имитационном моделировании изложены в [16, chapter 11]. Отметим, что наша задача, в отличие от классических моделей для систем с очередями, где анализируется, например, количество заявок в системе, обладает двумя особенностями, значительно облегчающими анализ выходных данных:

1) нет необходимости определять момент установления стационарного режима, все наблюдения анализируемой случайной величины — количества индексов для обслуживания запроса — являются одинаково репрезентативными;

2) все наблюдения случайной величины являются независимыми от предыдущих наблюдений, поэтому не требуется никаких дополнительных вычислений, чтобы отбросить влияние фактора автокорреляции.

Тем не менее при отладке модели требуется остановить выбор на достаточно четкой, хорошо апробированной и не требующей больших усилий при реализации процедуре определения момента завершения наблюдений (*terminating simulations*). В качестве таковой был выбран *метод групповых средних* (*batching means*). Метод был впервые подробно описан в [18], развит в работах тех же авторов [17] и [19]. Наиболее доступное его изложение со схемой реализации приведено в обзорной статье [24]. Следуя этой работе, приведем описание метода, реализованного в модели в функциях *Batching* (*Glo/Loc*), *Estimation* (*Glo/Loc*) и *SpectralVarAnalysis*, заодно исправив некоторые неточности, допущенные в оригинале (листинг 4). Итак, введем обозначения (их смысл становится более понятным из того, как они используются внутри метода):

M — количество групп, выбирается вычислителем, по умолчанию $M = 100$;

n_{max} — максимально возможное число наблюдений, не меньше M;

n_v — количество наблюдений, используемое для оценки дисперсии $\sigma^2[\overline{X}(n)]$, значение по умолчанию 100;

γ_a — коэффициент инкремента для контроля точности, значение по умолчанию 1,5;

(1-α) — уровень достоверности для итогового результата, значение по умолчанию 0,05;

ε_{max} — максимальное значение для относительной точности доверительного интервала, значение по умолчанию 0.05.

Листинг 4. Функции выходного анализа. Псевдокод

Listing 4. Pseudocode of output analysis procedures

```
Procedure OutputAnalysis
const M = 100;
procedure Batching:
{преобразование отдельных наблюдений в 2M групп с последовательным возрастанием размера группы}
begin
{вычисление среднего X̄j(m) на m наблюдениях и запись в j-й элемент буфера AnalysedSequence}
X̄j(m) := sum / m;
if j = 2M then
{укрупнение 2M групп размера m каждая в M групп размером 2m каждая}
for s:=1 to M do
X̄s(2m) := 0.5 (X̄2s-1(m) + X̄2s(m) );
enddo
m:=2m; j:=M;
endif
j:=j + 1; sum:=0;
{начало формирования следующей группы}
end Batching;
procedure Estimation:
{вычисляет оценку среднего и проверяет ее точность, до тех пор пока требуемая точность не будет достигнута}
begin
находим оценку дисперсии σ²[X̄(nv)] для последовательности из последних nv значений в AnalysedSequence и определяем степень свободы k (применением процедуры SpectralVarAnalysis);
вычисляем полуширину доверительного интервала для уровня достоверности (1 − α)
в текущей контрольной точке wk ε = t_{k,1-α/2} σ²[X̄(nv)] / X̄(jm), где X̄(jm) = Σ_{s=1}^{j} Xs(m) / j — текущее значение оцененного среднего после jm наблюдений, t_{k,1-α/2} — табличное значение распределения Стьюдента.
{проверяем условие прекращения моделирования}
if (ε ≤ εmax) then
{выводим результаты и заканчиваем моделирование}
StopSimulation:=true;
else
{требуемая точность еще не достигнута, устанавливаем следующую контрольную точку}
k:=k+1;
wk = min(⌊γa wk-1⌋, nmax);
end
end Estimation
begin {главная процедура, примерно соответствует функции cycle (Glo/Loc)}
```

Продолжение листинга 4

```
m:=1; {начальный размер группы}
k:=1; {номер первой контрольной точки}
w0 = 0; i:=1;
w1:=2M; {значение по умолчанию для первой контрольной точки}
sum:=0; j:=1;
StopSimulation:=false;
while (not StopSimulation) do
{собираем и обрабатываем новые w k-wk-1 наблюдений}
моделируем очередное значение/наблюдение x [wk-1+i], соответствует функции
getSingleData (Glo/Loc)
sum:=sum + x [wk-1 + i];
if (i mod m = 0) then Batching
endif;
if (i = wk) then Estimation
endif
if (not StopSimulation) then
i:=i + 1;
if (i > nmax) then
write («Решение с требуемой точностью не может быть получено»);
StopSimulation:=true;
endif
endif
enddo
write («За jm наблюдений с уровнем достоверности (1-α) получено решение
X̄(jm) (1 ± ε)")
end OutputAnalysis
procedure SpectralVarAnalysis
{Предусловия:
x (1), x (2), …, x (nv) — последовательность nv наблюдений стационарного процес-
са (nv ≥ 100)
nар. — количество точек усредненной периодограммы, используемых для приведения
в соответствие полиному применением метода наименьших квадратов (nар ≤ nv/4, зна-
чение по умолчанию 25);
δ — степень полинома, соответствующего логарифму усредненной периодограммы
(значение по умолчанию 2);
Cσ — нормализующая константа, выбранная так, чтобы px(0) было несмещенной оцен-
кой. Для значений по умолчанию nар. и δ Cσ=0.882 [17], где эти константы обозна-
чены k, d и C1 (k, d).
Шаг 1.
Вычислить 2 nар значений периодограммы для последовательности x (1), x (2), …, x
(nv)
П(j/nv) = |Σ_{s=1}^{nv} xs exp[−2πi(s − 1)j/nv]|² / nv для j =1, 2, …, 2 nар., i — корень из −1, мо-
дуль интерпретируется как модуль комплексного числа.
Шаг 2.
Вычислить nар. значений функции {L (fj)}, j = 1, 2, …, nар., где fj = (4j − 1)/2
nv и L (fj) = log { [П ( (2j − 1)/ nv) + П (2j/ nv)]/2}.
Шаг 3.
Методом наименьших квадратов находим коэффициент a0 полинома g(f) = Σ_{s=0}^{δ} as f^s, явля-
ющегося интерполяционным на множестве точек {L (fj) + 0.27}, j = 1, 2, …, nар.;
{a0 — это несмещенная оценка логарифма px (0)}.
```

Окончание листинга 4

```
Шаг 4.
Вычислить  p_x(0) = C_σ e^{-λ_1},  σ_ар^2[X̄(n_v)] = p_x(0) / n_v . Определяем k.
{таблица степеней свободы k для распределения хи-квадрат оценки σ_ар^2[X̄(n_v)] для
выбранных n_ар. и δ приведена в [17]; для n_v = 100, n_ар = 25 и δ = 2, k = 7}
end SpectralVarAnalysis
```

Рассмотренный метод анализа выходных данных достаточно прост и удобен в реализации и показал свою практическую работоспособность.

Апробация модели

Прежде чем перейти к описанию имитационных экспериментов и представлению их результатов, опишем метод верификации модели. Как уже упоминалось, в [10] построена аналитическая модель, работающая с двумя ограничениями — выполнением элементарного граничного условия и экспоненциальным $F_1(t)$. Поэтому имитационную модель можно считать достоверной, если при тех же входных данных она дает приемлемые по точности результаты в смысле совпадения с результатами аналитической модели. Многочисленные эксперименты показали, что это действительно так — расхождение в итоговом значении $Y_{ср.}$ составляет не более 0.1, относительная погрешность — не более 1%.

При выборе экспериментов акцент делался на показе преимуществ, которые дает имитационная модель по сравнению с аналитической, в частности, как отличаются результаты моделирования при выполнении ЭГУ и невыполнении такового. На рисунке 2 показаны зависимости $Y_{ср.}$ от k_1 для $N = 1$, $K = \{k_1\}$, $F_1(t) = 1 - e^{-\lambda t}$, $\lambda = 1/30$, $F_2(t) = 1 - e^{-\mu t}$, $\mu = 0.001$. Это соответствует появлению новой записи в БД примерно каждые 30 сек. и средней длине интервала запроса примерно 17 мин. Для нижней линии ЭГУ выполняется, т. е. $NB = \{3600\}$, $PB = \{1.0\}$, для верхней были выбраны параметры $NB = \{1,60,3600\}$, $PB = \{0.33, 0.33, 0.34\}$, т. е. начало запроса может с равной вероятностью приходиться на границу секунды, минуты или часа. Обратим внимание на принципиальное отличие случая выполнения ЭГУ и невыполнения такового. Если ЭГУ выполняется, то каждому множеству $K = \{k_1\}$ соответствует свое множество $NB = \{k_1\}$, что значительно снижает практическое значение результата, т. к. на практике выбирается ие-

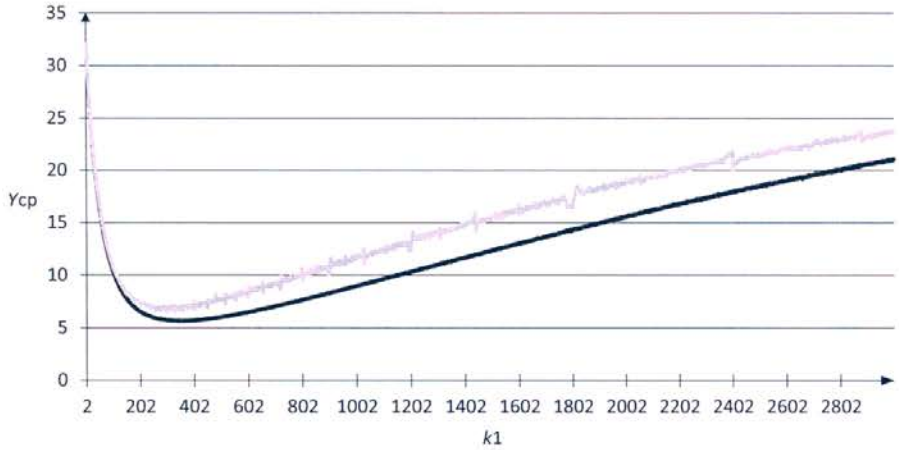

Рис. 2. $N = 1$. Зависимость $Y_{ср.}$ от $K = \{k_1\}$

Fig. 2. $N = 1$. Dependency average of Y on $K = \{k_1\}$

рархия, а множества NB и PB являются входными данными модели; если же требования выполнения ЭГУ нет, множество NB постоянно для каждого множества K. Данное обстоятельство объясняет две особенности полученного результата:

1) кривые практически совпадают при малых k_1 и все более удаляются друг от друга с его ростом. В самом деле чем больше k_1, тем дальше от левой границы интервала запроса находится левая граница ближайшего интервала для крупного индекса (при выполнении ЭГУ расстояние между этими левыми границами равно нулю), а следовательно, тем больше в начале интервала запроса нужно учесть мелких индексов. Поэтому с ростом k_1 расхождение между кривыми растет;

2) по той же причине наблюдаются флюктуации верхней кривой, т. к. на результат точечно влияет не только значение k_1, но и степень его «кратности» с границами крупных индексов.

Вместе с тем принципиальное поведение обеих зависимостей одно и то же. При выполнении ЭГУ минимум достигается при $k_1 = 354$ и равен 5.6, при невыполнении — примерно в диапазоне 320–340 и равен 6.5–6.6 (более точно сказать затруднительно, т. к. эти данные сильнее флюктуируют). С ростом k_1 $Y_{ср.}$ асимптотически стремится к предельному значению, соответствующему случаю, когда иерархии индексов нет. В [8] в предположении экспоненциальности обоих распределений для него была получена точная формула $Y_{ср.} = \frac{1-e^{-\lambda}}{e^{\mu}-1}$. При $\lambda = 1/30$, $\mu = 0.001$ эта формула дает значение 32.45. Из этого следует вывод о значительном увеличении производительности обработки запросов, которое дает использование иерархических индексов. На рисунке 3 те же две зависимости изображены для $N = 2$, $K = \{k_1, 3600/k_1\}$, т. е. для каждого k_1 в иерархию добавлен еще один крупный индекс, равный часу.

Эксперименты проводились только для делителей числа 3600. Нетрудно видеть некоторое улучшение результатов при добавлении второго уровня. Так, минимумы, достигаемые при $k_1 = 300$, равны 5.1 и 5.9, соответственно. Гладкость верхней линии в отличие от рис. 2 не должна удивлять, т. к. модель отработала только для делителей 3600, а соединение между ними выполнено простой линейной интерполяцией.

Рисунок 4 аналогичен рисунку 2 с той разницей, что для функций $F_1(t)$ и $F_2(t)$ принято распределение Вейбулла $F(t) = 1 - e^{-(\lambda t)^c}$. λ и c для $F_1(t)$ и $F_2(t)$ выбраны таким обра-

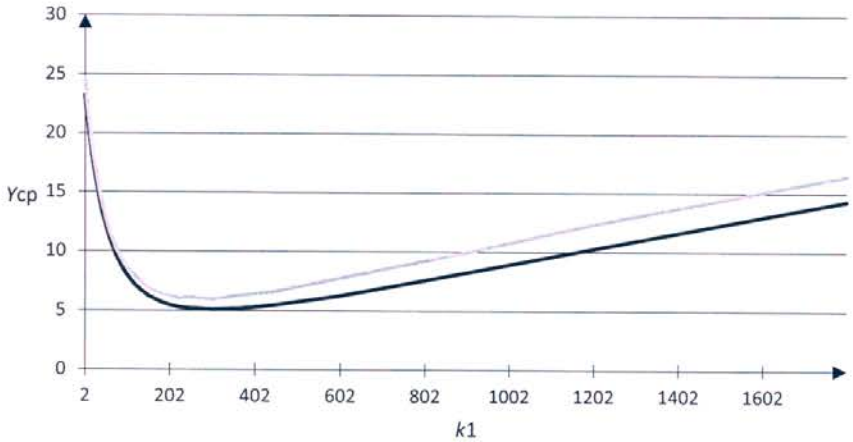

Рис. 3. $N = 2$. Зависимость $Y_{ср.}$ от $K = \{k_1, 3600/k1\}$

Fig. 3. $N = 2$. Dependency average of Y on $K = \{k_1, 3600/k_1\}$

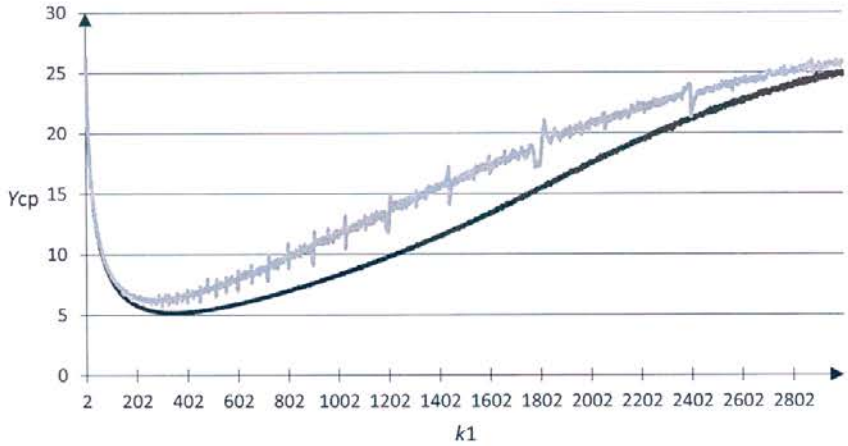

Рис. 4. $N = 1$. Распределение Вейбулла. Зависимость $Y_{ср.}$ от $K = \{k_1\}$

Fig. 4. $N = 1$. Weibull distribution. Dependency average of Y on $K = \{k_1\}$

зом, чтобы сохранить математическое ожидание соответствующих экспоненциальных распределений из рис. 2, что позволит увидеть влияние такого фактора как вид функции распределения. Для распределения Вейбулла первый момент, вычисляемый через гамма-функцию, равен

$$m_1 = \frac{\Gamma(1+\frac{1}{c})}{\lambda}.$$

Поэтому значения $c = 2$, $\lambda = 0.000887$ сохраняют для $F_2(t)$ среднее значение 1000; $c = 0.5$, $\lambda=1/15$ сохраняют для $F_1(t)$ среднее значение 30. Как известно [13], при $c > 1$ распределение Вейбулла является стареющим, при $c < 1$ — молодеющим, поэтому для моделирования были выбраны оба варианта.

Минимумы равны приблизительно 5.1 и 5.9, причем для модели без ЭГУ минимум достигается несколько раньше.

Заключение

Дальнейшее развитие построенной имитационной модели можно производить в сторону как систематизации, так и углубления. Создан инструмент исследования, и важно разработать стратегию его наиболее эффективного использования. Речь идет о планировании имитационных экспериментов для выявления особенностей и закономерностей, присущих моделируемой системе, т. к. ввиду трудностей ее теоретического изучения они вряд ли могут быть выявлены каким-либо иным способом. Факторов, которыми можно варьировать в модели и выявление влияния которых представляет интерес, здесь довольно много. К ним относятся:

- виды распределений $F_1(t)$ и $F_2(t)$ и параметры этих распределений;
- размер иерархии N;
- множество K, задающее непосредственно иерархию;
- множества NB и PB, регулирующие распределение начальной точки интервала запроса

Не исключено, что правильные вопросы, «заданные» модели, могут привести к весьма интересным и полезным в инженерном отношении ответам. Что же касается углубления модели, то и здесь то, что было описано в работе, можно рассматривать только как первый шаг в данном направлении. Помимо реализованного алгоритма существует еще множество алгоритмических вариаций применения иерархических bitmap-индексов, например:

1) динамический выбор между OR- и XOR-операциями на каждом уровне иерархии. Данная дисциплина описана и исследована в [14]. В ней решение принимается

на основании того, где фактически находится больше индексов — слева или справа от b_i;

2) использование стратегии реализации запроса, описанное в [22] как *cut selection*. Она позволяет предельно минимизировать число операций с индексами при обслуживании запроса, правда в описываемом случае — ненамного;

3) кэширование индексов. До сих пор предполагалось, что индексные битовые строки берутся откуда-то сами по себе, т. е. считываются с диска в память и вносят одинаковый вклад в трудоемкость обслуживания запроса. На самом деле таковую можно снизить, если какие-то из них сохранить в памяти и при повторном использовании брать готовыми из кэша. Вопрос, какие именно и от чего это может зависеть, ждет своего исследования.

Для всех описанных вариаций требуется расширение возможностей имитационной модели путем ее алгоритмической доработки. Кроме того, интерес могут представлять и такие задачи:

1) получение с помощью модели не только $Y_{ср.}$, но и распределения величины Y — количества индексов, участвующих в обслуживании запроса. Примеры графиков таких распределений приведены в [10];

2) нетрудно увидеть, что зависимости $Y_{ср.}$ от величины крупного индекса в целом однотипны (крутой спуск, затяжной минимум, где функция близка к константе, и далее медленный подъем с перегибом), поэтому вполне могут быть приближены каким-то параметризуемым семейством функций. Было бы целесообразно построить такую аппроксимацию. Заманчивые перспективы выполнения такого рода исследований открывает работа российского специалиста по имитационному моделированию [3];

3) наконец, фундаментальное решение поставленных в [10] задач оптимизации с помощью имитационного моделирования, содержательность которых проиллюстрирована данной работой, требует выбора соответствующей методики оптимизации. Пример такой методики дан в статье известного российского ученого [4].

Список литературы

1. *Артюхин В. В.* О некоторых особенностях проектирования и реализации имитационных моделей процессов в сложных технических системах // Прикладная информатика. 2011. №3 (33). С. 93–99.
2. *Боев В. Д.* Моделирование в AnyLogic. СПб.: Военная академия связи, 2016. — 412 с.
3. *Девятков Т. В.* Некоторые вопросы создания систем автоматизации имитационных исследований // Прикладная информатика. 2010. №5 (29). С. 102–116.
4. *Емельянов А. А.* Планирование экстремальных экспериментов с имитационными моделями // Прикладная информатика. 2013. №3 (45). С. 76–90.
5. *Карпов Ю.* Имитационное моделирование систем. Введение в моделирование с AnyLogic 5. СПб.: БХВ-Петербург, 2005. — 400 с.
6. *Кирстен В., Иринтер М., Кюн М.* СУБД Cache 5: объектно-ориентированная разработка приложений. 2-е изд. М.: Бином, 2005. — 416 с.
7. *Мартынов Д.* Bitmap-индексы в Cache на глобалах. URL: http://habrahabr.ru/company/intersystems/blog/174657/.
8. *Труб И. И.* Аналитическое вероятностное моделирование bitmap-индексов // Программные системы и вычислительные методы. 2016. №4. С. 315–323.
9. *Труб И. И.* О распределении количества bitmap-индексов для произвольного потока занесения записей в базу данных // Программные системы и вычислительные методы. 2017. №1. С. 11–21.
10. *Труб И. И.* Вероятностная модель иерархических индексов баз данных // Программные системы и вычислительные методы. 2017. №4. С. 15–31.
11. *Труб И. И.* Объектно-ориентированное моделирование на C++. СПб.: Питер, 2005. — 416 с.
12. *Труб И. И.* СУБД Cache: работа с объектами. М.: Диалог-МИФИ, 2006. — 480 с.
13. *Труб И. И., Труб Н. В.* Имитационное моделирование вычислительных систем. Учебный практикум. LAP Lampert Academic Publishing, 2015. — 344 с. URL: simulation.su/uploads/files/default/2015-uch-praktikum-trub-trub.pdf.
14. *Труб И. И., Труб Н. В.* Модель иерархических индексов баз данных с принятием решений и ее сравнение с минимаксной моделью // Программные системы и вычислительные методы. 2018. №1. С. 18–36.
15. *Шарма В.* Bitmap-индекс или B*tree-индекс: какой и когда применять? URL: http://citforum.ru/database/oracle/bb_indexes/.