УДК 519.256

Л. М. Волков

ХРОНОЛОГИЧЕСКИЕ ДЕРЕВЬЯ: СПОСОБЫ ПРЕДСТАВЛЕНИЯ В ПАМЯТИ*

1. Введение

Среди многочисленных, часто встречающихся на практике видов информации не последнее место занимают *хронологические структуры данных*[1]. Таким термином мы будем называть структуры данных, которые агрегируют в себе различные состояния (*срезы* или *версии*[2]) одной и той же базовой структуры данных, в которых она находится в различные моменты времени[3]. Если зафиксировать момент времени, то хронологическая структура данных превратится в одну из обычных. При этом в разные моменты времени состояния хронологической структуры могут различаться. Типичным примером хронологической структуры данных может служить список подразделений и сотрудников предприятия с информацией о вложенности подразделений и принадлежности сотрудника тому или иному подразделению. В задачах кадрового учета необходимо хранить всю историю изменений во времени списка

*Исследовательская работа финансировалась компанией «СКБ Контур».

[1] Терминологическое замечание: в англоязычной литературе [1] такие структуры данных называются *устойчивыми* (persistent). Однако, например, базы данных с зависимостью от времени принято все же называть именно *хронологическими* (chronological) или *временными* (temporal). Эти термины кажутся нам более естественными для обозначения структур данных, изменяющихся во времени и хранящих всю свою историю. Мы будем использовать первый из них (*хронологические структуры данных*).

[2] Обычно принято говорить о *версиях* хронологической структуры данных. Этот термин, на наш взгляд, является не вполне точным. Очевидно, что *версия* некоторого объекта в «житейском» смысле – это определенное, отличимое от других по некоторым признакам, состояние этого объекта. Однако, в смысле обсуждаемого ниже определения, версия хронологического дерева не является каким-то хронологическим деревом; она является просто деревом, т. е. абсолютно другим объектом! Термин *срез* хронологической структуры данных представляется нам поэтому гораздо более естественным для обозначения ее конкретного состояния в определенный момент времени.

[3] Везде далее мы полагаем, что время измеряется какими-то определенными квантами. Трактовка понятия «квант времени» зависит от конкретной прикладной задачи, в которой используется хронологическая структура данных. Где-то квантом времени является день, где-то секунда. Кванты времени нет нужды полагать одинаковыми. Например, во многих промышленных системах квантом времени может являться промежуток между двумя последовательными рабочими днями. Поэтому мы будем просто полагать, что все возможные моменты времени проиндексированы целыми числами.

подразделений и сотрудников; в то же время *срез* такого списка на каждый момент времени представляет собой, очевидно, дерево.

Настоящая статья является первой из трех, посвященных различным аспектам использования хронологических деревьев в прикладных задачах обработки больших массивов изменяющихся во времени данных. Общая цель серии статей заключается в том, чтобы предложить практическое решение для ряда так или иначе связанных между собой вопросов, имеющих отношение к хронологическим деревьям и возникших в ходе разработки инструментальной среды, предназначенной для быстрого создания приложений, ориентированных на работу именно с такими данными в прикладных задачах. Каждая из статей концентрируется в основном на одной ключевой задаче работы с хронологическими деревьями, возникшей из практики разработки системы. В фокусе внимания данной статьи – задача нахождения такого способа представления хронологического дерева в оперативной памяти, который обеспечил бы, с одной стороны, быстрое и удобное выполнение операций навигации и, с другой стороны, быструю и однозначную упаковку дерева в реляционную базу данных и восстановление дерева из такой базы. Помимо этого в первой статье цикла вводится весь необходимый понятийный аппарат. Вторая статья будет посвящена целиком задаче проверки целостности хронологического дерева при внесении в него единичных и массовых изменений (в терминах практической задачи – решается проблема разработки менеджера транзакций для хранилища данных, построенного на основе хронологического дерева). В третьей статье будут предложены подходы к корректной обработке ситуаций, связанных с изменением реального носителя хронологического дерева в процессе работы с ним.

Дадим основные определения. Пусть T – это некоторый абстрактный тип данных (мы далее будем пользоваться общепринятой аббревиатурой – АТД). У каждой структуры данных типа T есть свой *носитель* – множество элементов данных. Например, носителем стека или очереди является линейный список их элементов, а, соответственно, *структуру* конкретного АТД – «стек» или «очередь» задают конкретные методы доступа к данным и связи между элементами.

Типом данных «хронологический(ое, ая) T» мы будем называть отображение, которое каждому моменту времени из некоторого интервала ставит в соответствие структуру данных типа T, при этом для всех моментов времени носители структур, которые порождаются отображением, совпадают. Носитель хронологической структуры данных определяется тогда естественным образом как носитель каждой из структур данных, которая является срезом хронологической структуры на определенный момент времени. В этих терминах набор всех сотрудников и подразделений предприятия представляет

собой носитель для *хронологического дерева* предприятия[4]. Хронологическое дерево содержит в себе всю информацию обо всех изменениях в структуре предприятия. Там, например, может содержаться информация о том, что И. И. Иванов с первого июня 2002 г. работает в бухгалтерии, а с первого июля – в отделе кадров. С другой стороны, в каждый конкретный момент времени хронологическое дерево посредством перехода к срезу позволяет получить вполне определенное дерево структуры предприятия. И если во многих задачах обработки и хранения информации достаточно хранить только актуальное состояние данных, то столь же во многих встает проблема хранения всех срезов данных, т. е. хронологических структур данных.

В литературе [2, 3] проводится естественное разделение между *частично хронологическими* (partially persistent) и *полностью хронологическими* (fully persistent) структурами данных. Первые допускают только чтение исторических состояний, а вторые – также и редактирование. В большинстве прикладных задач встречаются только частично хронологические структуры данных. Однако ниже мы, не прилагая дополнительных усилий, сделаем наши структуры данных полностью хронологическими без каких-либо потерь в памяти и производительности. Поэтому мы говорим просто о хронологических структурах, не разделяя их на частично или полностью хронологические.

2. Постановка задачи

2.1. Операции

При описании любого АТД главной задачей является четкое определение операций над данными, которые должен поддерживать этот АТД. Именно различие операций доступа к данным определяет различие между такими АТД, как уже упоминавшиеся стек и очередь, работающими на одинаковом носителе. Существует достаточно много реализаций АТД «дерево», незначительно различающихся поддерживаемыми операциями. Естественно, выбор той или иной реализации в конкретной задаче определяется требованиями этой задачи. Абсолютно лучшей реализации дерева не существует: в каждой реализации некоторые операции работают быстрее, а некоторые медленнее.

[4] В зависимости от конкретного приложения может иметь смысл рассматривать объекты с временем жизни, ограниченным снизу или сверху (например, у тех же сотрудников предприятия всегда известна дата принятия на работу и иногда – дата увольнения). Тем не менее, как правило, полагают, что все срезы определены на одном и том же носителе. Для работы с вершинами, которые в данном срезе «отсутствуют», вводят специальную «фиктивную» вершину, которую назначают в родители «отсутствующим» объектам. Иногда приходится вводить даже две специальные вершины – чтобы различать «еще не родившиеся» и «уже умершие» объекты.

Соответственно, то же самое справедливо и для хронологических деревьев. Ниже мы имеем дело не с хронологическими деревьями «вообще», а с хронологическими деревьями, которые должны эффективно реализовывать определенный набор операций. Этот набор операций, однако, является весьма естественным и возникает во многих практических задачах. Помимо ограничения по набору операций, мы вводим еще ограничение по структуре хранимых данных. А именно предполагается, что в практических задачах хронологические деревья являются достаточно *сбалансированными*. Мы считаем, что каждое из деревьев, представляющее собой срез хронологического дерева на определенный момент времени, имеет высоту $h = O(\log N)$, где N – мощность носителя (количество объектов в дереве). Это допущение очевидным образом следовало из природы данных в практических задачах, инициировавших наше исследование. Впрочем, все сказанное ниже можно будет распространить и на хронологические самобалансирующиеся деревья (например, B-деревья или 2–3-деревья): их можно будет представлять с помощью аналогичной хронологической структуры данных, и точно так же будет работать основная операция добавления новой связи – только после каждого добавления в соответствующих срезах понадобится балансировка.

Далее будут закреплены следующие обозначения:

N – количество объектов в хронологическом дереве;

K – максимальная длина истории изменений объекта;

ObjectID, OID – уникальный идентификатор объекта;

ParentID, PID – уникальный идентификатор родителя;

LCID – уникальный идентификатор крайнего слева ребенка;

RBID – уникальный идентификатор ближайшего справа брата;

NULL – нулевой идентификатор, обозначающий отсутствие объекта;

Timestamp – момент времени;

Start – момент начала действия связи;

Ch – используется в названиях операций над хронологическими деревьями, чтобы отличать их от аналогичных операций над обычными деревьями.

Мы будем предполагать, что множество всех объектов (т. е. носитель) линейно упорядочено каким-либо образом. Без ограничения общности можно, например, полагать объекты упорядоченными по возрастанию OID. Семантически это упорядочение не несет никакой нагрузки, а используется лишь для того, чтобы однозначно определять порядок слева направо. В частности, для каждого объекта в дереве благодаря этому порядку мы будем однозначно определять крайнего слева ребенка и ближайшего справа брата. Для краткости ниже мы будем писать «левый ребенок» и «правый брат» вместо соответственно «крайний слева ребенок» и «ближайший справа брат».

Для записи алгоритмов используется язык, близкий к языку C++. Как

это обычно принято, мы опускаем операторы, связанные с приведением типов, работой со стандартными структурами данных и т. п. При записи алгоритмов также предполагается, что реализован класс ChTree, представляющий собой хронологическое дерево, и класс Tree, реализующий обычное дерево.

Итак, далее рассматривается АТД «хронологическое дерево», поддерживающий следующие операции:

GetChParent(ObjectID, Timestamp) – возвращает идентификатор родителя узла по идентификатору узла и моменту времени;

GetChLChild(ObjectID, Timestamp) – возвращает идентификатор левого ребенка узла по идентификатору узла и моменту времени;

GetChRBrother(ObjectID, Timestamp) – возвращает идентификатор правого брата узла по идентификатору узла и моменту времени;

GetCut(Timestamp) – возвращает дерево, которое является срезом хронологического дерева на данный момент времени;

AddLink(ObjectID, ParentID, Start) – добавляет к существующему целостному хронологическому дереву *связь* – информацию о том, что объект ParentID является родителем объекта ObjectID начиная с момента времени Start.

Указанные операции, будучи реализованными, дают возможность:

1) осуществлять любую навигацию по хронологическому дереву;

2) получать срезы дерева на любой момент времени;

3) инициализировать хронологическое дерево за один проход из списка связей, хранящегося в реляционной таблице в произвольном порядке.

Остановимся теперь на этих возможностях более подробно для того, чтобы показать область применимости хронологического дерева, поддерживающего такие операции.

2.2. Навигация

Нетрудно понять, что, как и для обычного дерева, для хронологического дерева достаточно операций перехода к родителю, левому ребенку и правому брату для полной и удобной навигации по структуре данных. С помощью этих базовых операций можно моделировать все нужные на практике методы навигации и доступа к множествам узлов хронологического дерева. В качестве примера рассмотрим три наиболее важных процедуры навигации.

1. Получение списка всех детей узла на момент времени:

```
ChTree::GetChAllChildren(OID, Timestamp) {
TempID = OID;
```

```
ListOfChildren = new Stack();
TempID = ChTree.GetChLChild(TempID, Timestamp);
while (TempID!=NULL) {
ListOfChildren ⇐ TempID;
TempId = ChTree.GetChRBrother(TempID, Timestamp);
}
return ListOfChildren;
}
```

2. Получение корня того дерева, в котором находится узел на данный момент времени:

```
ChTree::GetChRoot(OID, Timestamp) {
TempID = OID;
while (ChTree.GetChParent(TempID, TimeStamp)!=NULL)
TempId = ChTree.GetChParent(TempID, Timestamp);
return TempId;
}
```

3. Получение последовательности вершин хронологического дерева при обходе в глубину на данный момент времени (эта достаточно сложная операция часто нужна на практике, например, при заполнении визуальных элементов управления):

```
ChTree::DFS(OID, Timestamp) {
Sequence ⇐ OID;
TempId = ChTree.GetChLChild(OID, Timestamp);
if (TempId!=NULL)
ChTree.DFS(TempID, TimeStamp);
TempId = ChTree.GetChRBrother(OID, Timestamp);
if (TempId!=NULL)
ChTree.DFS(TempID, TimeStamp);
return;
}
ChTree::GetChDFSSequence(Timestamp) {
Sequence = new Stack();
while («есть непросмотренные корни») {
TempID = «очередной корень»;
ChTree.DFS(TempID, Timestamp);
}
return Sequence;
}
```

Итак, очевидно, что все операции навигации по дереву для хронологических деревьев можно реализовать полностью аналогично операциям навигации для обычных деревьев. Более того, можно предложить такую реализацию трех базовых операций навигации:

```
ChTree::GetChParent(OID, Timestamp) {
Tree = ChTree.GetCut(TimeStamp);
PID = Tree.GetParent(OID);
return PID;
}
ChTree::GetChLChild(OID, Timestamp) {
Tree = ChTree.GetCut(TimeStamp);
LCID = Tree.GetLChild(OID);
return LCID;
}
ChTree::GetChRBrother(OID, Timestamp) {
Tree = ChTree.GetCut(TimeStamp);
RBID = Tree.GetRBrother(OID);
return RBID;
}
```

Однако, как мы покажем ниже, подобные «прямолинейные» решения оказываются чрезвычайно неэффективными; можно реализовать базовые операции навигации (а следовательно, и все возможные операции навигации) гораздо удачнее, нежели непосредственным переходом к обычным деревьям. Оказывается, что можно использовать специфику хронологических деревьев для эффективной реализации операций навигации. Более точно, построение среза само по себе требует $O(N \cdot \log K)$ операций, а мы укажем способ выполнения базовых операций навигации по хронологическому дереву за время $O(\log K)$.

2.3. Срезы

Операция перехода к срезу является одной из двух основных операций для хронологических деревьев, которая не имеет аналогов для обычных деревьев. На практике эта операция оказывается часто основной операцией над хронологическими деревьями. Хотя, как мы и отмечали выше, базовые операции навигации эффективнее выполнять без перехода к срезу, ситуация меняется для многих сложных операций. Так, например, построение обхода в глубину для данного момента времени оказывается дешевле по временным затратам, если осуществлять его на предварительно построенном срезе. Поэтому хронологическое дерево, которое мы реализуем, обязано обеспечить максимально

производительное выполнение операции перехода к срезу по указанному моменту времени.

2.4. Инициализация

В настоящей работе мы решаем задачу представления данных хронологического дерева в виде структуры в памяти. Эта структура не является статической; наиболее типичной операцией с хронологическим деревом является добавление новой связи, дата начала действия которой не меньше максимальной даты, определенной в дереве к настоящему моменту времени. Эта операция соответствует случаю, когда в прикладной задаче ведется работа с данными хронологического дерева: удаление или изменение уже существующих «исторических» связей не допускается, а все изменения вносятся начиная с текущей даты.

Мы будем рассматривать операцию добавления в существующее хронологическое дерево произвольной связи, заданной тройкой: (ObjectID, ParentID, Start). Таким образом, мы решим не только задачу добавления одной связи, но и задачу начального заполнения пустого хронологического дерева из таблицы реляционной базы данных, хранящей произвольное неупорядоченное множество таких троек. Мы будем полагать, что данные, хранящиеся в таблице, которая используется для инициализации хронологического дерева, удовлетворяют естественным ограничениям целостности[5]:

1) в произвольный момент времени срез хронологического дерева представляет собой ациклическую структуру;

2) в каждый момент времени, если для некоторого объекта определен родитель, то либо для этого родителя также определен родитель, либо он является корнем.

Если это так, то, последовательно просматривая записи (ObjectID, ParentID, Start), мы за один проход по таблице восстановим хронологическое дерево целиком. Это имеет большое практическое значение, поскольку указывает, например, весьма удобный способ хранения данных из структуры в памяти на диске между сеансами работы с хронологическим деревом и быстрой загрузки их в оперативную память при необходимости.

В настоящей работе мы не рассматриваем поддержку целостности для хронологических деревьев. Эта отдельная, широкая и интересная тема (как проверить целостность данных в хронологическом дереве, как эффективно

[5] Как отмечалось во введении, задача обеспечения целостности сама по себе имеет очень большое практическое значение и рассматривается в отдельной работе; здесь же мы предполагаем, что эта задача уже решена.

убедиться в том, что поступающее изменение или группа одновременно поступающих изменений не нарушит целостности) является предметом исследования второй статьи цикла.

Еще одно важное замечание заключается в том, что предложенный ниже алгоритм добавления в целостное хронологическое дерево ровно одной связи на практике ценен сам по себе только при решении задачи восстановления дерева из списка связей. Если же говорить про оперативное пополнение дерева в оперативной памяти, то этот алгоритм будет востребован только в составе более сложной системы – менеджера транзакций, способного обрабатывать одновременные запросы многих пользователей на внесение новых связей. Алгоритм работы менеджера транзакций предложен и обоснован во второй статье.

3. Реализация хронологического дерева

3.1. Организация данных

Для реализации хронологического дерева, поддерживающего заявленные выше методы и обладающего высокой производительностью, мы предлагаем использовать хронологическую модификацию известной структуры «левый ребенок – правый брат», которая широко применяется для представления в памяти обычных деревьев. Напомним, что эта структура состоит из трех массивов Parent, LChild и RBrother, которые индексированы идентификаторами объектов и содержат ссылки на родителя, левого ребенка и правого брата каждого узла соответственно. Реализация базовых операций навигации для такой структуры данных очевидна.

Будем теперь более широко трактовать массивы Parent, LChild и RBrother; посмотрим на них как на *отображения,* которые ставят в соответствие ключу – идентификатору объекта – значение: идентификатор родителя, левого ребенка и правого брата соответственно. Отображение само по себе является популярным АТД; в языке C++ оно реализовано в виде контейнерного класса map. Используя любой язык программирования, нетрудно реализовать АТД «отображение» (поддерживающий очевидные методы добавления, удаления и извлечения пары ключ–значение) с помощью, например, хэш-таблицы. Визуально мы можем представить эти отображения так:

Tree := { OID ⇒ PID,
OID ⇒ LCID,
OID ⇒ RBID };

АТД «дерево» можно эффективно реализовать с помощью трех экземпляров АТД «отображение».

Используя введенную нотацию, предложим теперь эффективную реализацию хронологического дерева:

ChTree := { OID ⇒ (Start ⇒ PID),

OID ⇒ (Start ⇒ LCID),

OID ⇒ (Start ⇒ RBID) };

Отображение (Start ⇒ PID) хранит в себе *историю* изменений указателя на родителя данного объекта и представляет собой упорядоченный по ключу Start набор значений идентификатора родителя PID для моментов времени Start. Например, история из трех записей

(01.2002 ⇒ 5,

04.2002 ⇒ NULL,

09.2002 ⇒ 7)

обозначает, что данный объект до января[6] не был определен (если в системе не приняты некоторые умолчания; например, на практике можно полагать, что неопределенный объект является изолированным одноэлементным деревом, т. е. что значения PID, LCID и RBID по умолчанию равны NULL), с января по март его родительским элементом был объект с идентификатором 5, с апреля по август данный объект был корнем некоторого дерева, а с сентября и по настоящий момент времени его родителем является объект с идентификатором 7. Заметим, что мы здесь предположили, что система оперирует с открытыми справа полуинтервалами времени; конечно же, это тоже вопрос договоренности, хотя использование именно таких полуинтервалов и представляется наиболее естественным.

История изменений указателей на левого ребенка и правого брата имеет аналогичный смысл. Ранее мы предполагали, что длины всех историй изменения всех указателей всех объектов системы равномерно ограничены некоторой небольшой константой K. На практике это обычно имеет место (например, если рассматривается задача кадрового учета с подразделениями и сотрудниками в качестве элементов данных хронологического дерева, то длина истории изменения указателя на родителя отражает количество переходов сотрудника из отдела в отдел). Впрочем, как мы увидим, в нашей реализации хронологического дерева зависимость времени выполнения любой операции от K окажется логарифмической, поэтому длина истории изменения указателя не представляет, вообще говоря, особой практической важности.

Итак, мы предлагаем реализацию хронологического дерева в виде трех отображений, первое из которых ставит в соответствие объекту историю изменений его указателя на родителя, а второе и третье – то же самое для левого ребенка и правого брата. Теперь мы должны описать эффективную

[6] В этом примере для простоты квантом времени считается месяц.

реализацию заявленных пяти операций работы с хронологическим деревом на такой структуре данных.

3.2. Навигация

Реализовать три базовых операции навигации, используя структуру данных на трех отображениях, совсем несложно. В качестве примера рассмотрим переход к родителю. Для того чтобы найти родителя данного объекта на момент времени Timestamp, достаточно всего лишь осуществить двоичный поиск наибольшего ключа, не превосходящего Timestamp, в истории изменения указателя на родителя этого объекта. Такой поиск (кстати, контейнерный класс map языка C++ предоставляет программисту готовую реализацию двоичного поиска) отрабатывает, очевидно, за время $O(\log K)$. Абсолютно аналогично реализуются операции перехода к левому ребенку и правому брату.

3.3. Срезы

Процедура построения среза также реализуется тривиально. Для получения дерева, являющегося срезом хронологического дерева на момент времени Timestamp, заданного стандартным образом в виде массива Parent, достаточно просмотреть в цикле все элементы хронологического дерева и найти родителей на момент времени Timestamp. Если есть необходимость получения дерева, заданного тройкой отображений (родитель – левый ребенок – правый брат), также достаточно одного прохода с нахождением соответственно родителя, левого ребенка и правого брата на данный момент времени. Итак, срез может быть построен за время $O(N \cdot \log K)$.

3.4. Инициализация

Наиболее сложной задачей является инициализация рассматриваемой структуры за один проход из списка связей. Для того чтобы решить эту задачу, надо, естественно, только научиться эффективно добавлять одну связь в уже существующее целостное хронологическое дерево.

Итак, на вход алгоритма поступает тройка: (OID, PID, Start) и, используя указанные в ней данные, необходимо обновить три массива (истории): OID.ParentID, PID.LCID и OID.RBID. Соответственно, нужно указать способ для пересчета каждого из этих массивов. Ясно, что другие истории напрямую добавлением новой связи не затрагиваются.

Пересчет массива OID.ParentID. Эта задача также решается просто. В массив добавляется новый элемент истории (Start $\Rightarrow$ PID), который указывает, что теперь с момента времени Start родителем объекта OID является объект PID. При этом если уже есть запись в истории типа ($t_1 \Rightarrow \mathrm{PID}_1$), где $t_1 >$Start, то автоматически получится, что вносимое изменение имеет смысл

только на полуинтервале [Start, t_1), где t_1 – наименьший такой момент времени. Если же ни одной записи ($t_1 \Rightarrow \mathrm{PID}_1$), где $t_1 >$Start, в истории объекта OID не нашлось, то автоматически будет считаться, что объект PID является родителем объекта OID, начиная с момента времени Start и до бесконечности (или по настоящий момент времени в зависимости от трактовки конкретной прикладной задачи).

Такая операция вставки может быть осуществлена (при правильной реализации ассоциативного массива истории в виде сортирующего дерева) за время $O(\log K)$. Попутно заметим следующее: если в результате в списке истории окажется две записи с одним значением времени Start и разными значениями PID, то таким образом целостность данных будет нарушена; ведь система не будет способна как-то однозначно определить, какой же объект является родителем OID начиная с момента времени Start; мы не можем делать никаких предположений о том, как внутри системы упорядочиваются события с одинаковой датой начала, и можем только считать этот порядок абсолютно произвольным.

Пересчет массива PID.LCID. Нетрудно понять, что указатель на левого ребенка объекта PID может изменяться на всем интервале [Start, t_1), где метка времени t_1 имеет описанное выше значение, даже если непосредственно в момент времени Start объект OID не становится самым левым ребенком объекта PID. В частности, если записи истории с $t_1 >$ Start не существует, то проверять изменения указателя на левого ребенка необходимо с настоящего момента и до бесконечности.

Пусть история PID.LCID имеет вид

$$\{s_i \Rightarrow \mathrm{LC}_i\}.$$

Пусть также j – максимальный индекс записи этой истории, такой, что $s_j <=$ Start, а k – максимальный индекс, такой, что $s_k < t_1$. Тогда, чтобы правильно преобразовать массив PID.LCID, необходимо сравнить OID с идентификатором левого сына LC_j, LC_{j+1}, …, LC_k на соответствующих интервалах времени (Start, s_{j+1}), $(s_{j+1}, s_{j+2}), \ldots, (s_k, t_1)$. На каждом из этих интервалов, если выполняется неравенство OID $< \mathrm{LC}_i$, то объект OID замещает собой самого левого сына. Технически при $j < i < k$ это делается с помощью замены элемента истории ($s_i \Rightarrow \mathrm{LC}_i$) на ($s_i \Rightarrow$ OID). В двух крайних случаях изменения в истории PID.LCID немного сложнее:

1) если $i = j$, то надо добавить в историю запись (Start $\Rightarrow$ OID);

2) если $i = k$, то надо заменить запись ($s_k \Rightarrow \mathrm{LC}_k$) на ($s_k \Rightarrow$ OID) и добавить в историю запись ($t_1 \Rightarrow \mathrm{LC}_k$).

Пересчет массива OID.RBID. Указатель на правого брата объекта OID также может изменяться на любом подынтервале интервала [Start, t_1). При этом a priori указать возможные границы для подынтервалов, на которых значение указателя на правого брата может измениться, невозможно. Если в предыдущем случае мы могли сразу определить моменты времени s_i, в которые надо производить сравнение OID и текущего указателя, то здесь эти моменты времени еще придется вычислять. Последовательность действий, которые надо проделать, чтобы обновить историю изменений указателя на правого брата для объекта OID, выглядит следующим образом:

Шаг 1. Положим $S = \text{Start}$.

Шаг 2. Если $S \geq t_1$ (или если $t_1 > \text{Start}$ не существует и $S = \infty$), то выход. Иначе для момента времени S вычислим левого ребенка объекта PID, обозначим его как C_1. Пусть также связь, определяющая, что C_1 – это левый ребенок PID в момент времени S, действует до момента времени s_1(как обычно, может быть так, что $s_1 = \infty$). Положим $i = 1$.

Шаг 3. Если $C_i = \text{NULL}$, то перейти к следующему шагу. Иначе вычислим C_{i+1} как правого брата объекта C_i в момент времени S. Вычислим также s_i как момент времени, до которого действует связь, определяющая, что C_{i+1}– это правый брат объекта C_i в момент времени S.

Шаг 4. Теперь построен список $C_1, \ldots C_k$ детей объекта PID на момент времени S. Вычислим $T = \min\{s_1, \ldots, s_k\}$. Тогда, по построению, $T > S$ и T – это момент времени, до которого список детей объекта PID, построенный на момент времени S, остается неизменным (возможно, что $T = \infty$).

Шаг 5. Найдем такой номер ребенка m, что $C_m < \text{OID} < C_{m+1}$. Перестроим истории изменений правого брата для объектов C_m и OID так, чтобы с момента времени S по момент времени T правым братом объекта C_m был бы OID (это надо сделать, если $m > 0$), а правым братом объекта OID был бы C_{m+1} (при этом возможно, что C_{m+1}=NULL). Технически для этого понадобится добавление в историю следующих записей:

– в массив C_m.RBID: $(S \Rightarrow \text{OID})$ и $(T \Rightarrow C_{m+1})$, причем последняя из этих записей может добавляться, только если $T < \infty$;

– в массив OID.RBID: $(S \Rightarrow C_{m+1})$.

Запись вида $(T \Rightarrow \text{NULL})$ специально добавлять не надо, поскольку на момент времени T пересчет еще будет производиться в последующем. Запись $(T \Rightarrow C_{m+1})$ надо добавить только тогда, когда в массиве C_m.RBID еще нет записи с ключом T.

Шаг 6. Положим $S = T$. Перейдем ко второму шагу.

Аналогичным образом может быть реализован пересчет соответствующих массивов и для задач изменения связи или удаления связи из полностью хронологического дерева.

4. Частично хронологическое дерево

Как мы уже отмечали выше, на практике важным случаем является *частично хронологическое* дерево, которое позволяет добавлять только связи, время начала действия которых совпадает с текущим моментом времени; удаление и изменение уже добавленных связей не допускается. При этом операции доступа к данным для частично хронологического дерева не отличаются от рассмотренных выше.

Таким образом, для частично хронологического дерева достаточно рассматривать только упрощенный аналог приведенного выше алгоритма добавления произвольной связи к целостному хронологическому дереву – алгоритм добавления связи, с временем Start, которое не меньше максимального времени Start всех уже присутствующих в дереве связей (алгоритм добавления связей «в конец» хронологического дерева). Такой алгоритм становится тривиальным. Пересчет массива OID.ParentID производится точно так же, как и в алгоритме добавления произвольной связи, в алгоритмах пересчета массивов PID.LCID и OID.RBID цикл по временным меткам заменяется единственной операцией сравнения в момент времени Start. Таким образом, для этого случая легко удается провести оценку временной сложности алгоритма добавления связи. Ясно, что пересчет всех массивов, кроме OID.RBID, требует константного времени. В то же время обновление массива OID.RBID неизбежно требует построения списка всех детей объекта PID, которое, в худшем случае, может занять время порядка $O(N)$. На практике, однако, чаще встречаются деревья, где максимальное количество детей узла ограничено некоторой константой C, которая существенно меньше, чем N.

Теперь для инициализации частично хронологического дерева можно предложить сначала проводить сортировку всего списка связей по времени начала действия связи, а потом применять описанный только что алгоритм добавления связей «в конец». Тогда, поскольку общее количество связей не превосходит $N \cdot K$, общее время, требуемое для инициализации хронологического дерева, составит $O(C \cdot N \cdot K + N \cdot K \cdot \log(N \cdot K))$. В зависимости от конкретной практической ситуации окажется реально возможным пренебречь одним из этих слагаемых; чаще все же второе слагаемое оказывается больше, чем первое. На практике также выполняется неравенство $K < N$, и потому вычислительная сложность такого алгоритма составляет, реально, $O(N \cdot K \cdot \log N)$.

Для полностью хронологического дерева, однако, алгоритм добавления связи «на произвольное время» должен быть реализован в любом случае, поэтому при инициализации полностью хронологического дерева у нас есть выбор: использовать алгоритм добавления связи «в конец» с предваритель-

ной сортировкой или отказаться от сортировки. В этом случае вычислительная сложность инициализации составит $O(N \cdot K \cdot T)$, где T – среднее время работы алгоритма добавления одной связи. Можно понять, что в наших ограничениях $T = O(N \cdot K)$ (не более чем K записей в истории изменения указателя на правого брата и не более чем N детей объекта PID просматривается каждый раз, когда работает цикл по времени в алгоритме обновления массива OID.RBID; все остальные массивы истории обновляются существенно быстрее). Однако в вычислительном эксперименте T – не очень большая константа[7] и, как правило, алгоритм инициализации без предварительной сортировки работает даже быстрее, чем алгоритм с предварительной сортировкой записей по времени с последующим добавлением записей «в конец». В целом можно резюмировать, что выбор того или иного алгоритма инициализации определяется ограничениями конкретной прикладной задачи. Во встречавшихся нам задачах эти методы оказывались приблизительно равноценными.

5. Заключительные замечания

5.1. Что еще можно сделать

Теоретически можно пытаться довести стоимость выполнения базовых операций навигации в лучшем случае до $O(1)$, а стоимость построения среза – соответственно, до $O(N)$. Возможно, что на промежутке от $O(1)$ до $O(\log K)$ скрыты возможности для построения реализаций хронологических деревьев, удовлетворительных по объему используемой памяти и более производительных, нежели реализация, предлагаемая в данной работе. На самом деле то, что было рассмотрено выше, является приложением хорошо известного (см. основополагающую работу [1]) *метода насыщения узла* (fat node method), который заключается именно в замене каждой ссылки статической структуры данных историей изменения ссылок в хронологической структуре. Поэтому основная ценность данной работы может заключаться не в рассмотрении самой структуры данных (по сути дела, оно носило вводный характер), а в предложенном алгоритме инициализации хронологического дерева из реляционной структуры.

Что же касается методов перехода от обычных структур данных к хронологическим, то, помимо метода насыщения узла, в литературе (например, в той же работе [1]) описывается еще и *метод копирования узла* (node copying

[7] Мы пока не можем подкрепить результаты эксперимента выкладками, однако на практике количество операций T всегда так мало, что есть ощутимая надежда доказать тот факт, что в среднем T – константа.

method). Этот метод действительно более требователен к памяти, и решает базовые задачи навигации с амортизационной стоимостью $O(1)$. Методом копирования узла можно автоматически получать хронологические аналоги тех структур данных, в которых каждому элементу данных соответствует единственный указатель (стеки, очереди, деревья, реализуемые только массивом PARENT). Однако реализация методом копирования узла даже таких простых двухсвязных структур, как хронологические деки, оказалась весьма затруднительной и потребовала серьезных исследований (см. [3, 4]). Пока мы не готовы предложить эффективную реализацию метода копирования узла для рассматриваемых нами «трехсвязных» деревьев.

5.2. Еще раз о практическом смысле

Итак, выше была описана структура данных, реализующая одну из версий АТД «хронологическое дерево». Особый акцент был сделан на алгоритм инициализации этой структуры данных из списка связей, который мы и считаем основным результатом работы. Каким же может быть практическое применение этого алгоритма? Этот алгоритм фактически позволяет эффективно организовать хранение хронологических деревьев в обычных реляционных базах данных. Используя реляционную таблицу для хранения всей истории изменения связей между сеансами работы некоторого приложения, работающего с хронологическим деревом, мы получаем возможность эффективно восстанавливать требуемую структуру в оперативной памяти в начале сеанса работы и не менее эффективно сохранять хронологическое дерево в реляционной таблице в конце сеанса.

Предложенная схема работы с хронологическими деревьями была апробирована на практике и хорошо себя зарекомендовала. Так, описанный алгоритм хранения и восстановления полностью хронологических деревьев по реляционной таблице связей применяется в клиент-серверной системе «Контур-Персонал», разработанной екатеринбургской компанией «СКБ Контур». Система «Контур-Персонал» внедрена и работает на ряде крупных предприятий Уральского региона; на крупнейшем из них численность работающих достигает 30 тыс. человек.

Автор отдельно хотел бы обратить внимание на то, что данная теоретическая работа была бы невозможной без усилий постановщиков «СКБ Контур», обнаруживших проблему и обративших на нее внимание автора, и программистов этой компании, реализовавших предложенные автором структуры данных в промышленном проекте, на практике подтвердив их применимость. Особую благодарность автор хотел бы выразить руководителю разработчиков компании «СКБ Контур» Эдуарду Романовичу Шифману.

Литература

1. Driscoll J. R., Sarnak N., Sleator D. D., Tarjan R. E. Making data structures persistent // J. Comput. System. Sci. 1989. Vol. 28, № 1. P. 86–124.

2. Dietz P. F. Fully persistent arrays // Lect. Notes Comp. Sci. 1989. Vol. 382. P. 67–74.

3. Driscoll J. R., Sleator D. D., Tarjan R. E. Fully persistent lists with catenation // Journal of the ACM. 1994. Vol. 41, № 5. P. 943–959.

4. Kaplan H., Okasaki C., Tarjan R. E. Simple confluently persistent catenable lists // SIAM Journal on Computations. 2000. Vol. 30, № 3. P. 965–977.