ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Автономная некоммерческая организация науки и образования

«Институт компьютинга»

УТВЕРЖДАЮ

Директор АНО «Институт компьютинга»

_____ /А.И. Миков/

«_ _»_____2008 г.

м.п.

# РЕКЛАМНО-ТЕХНИЧЕСКОЕ ОПИСАНИЕ

Средства тиражирования данных METAS Replication

.31569113.00006-01 99 01

Листов 55

Разработчики:

_____/Стрелков М.А./

_____/ Шаврина О.В./

_____/ Лядова Л.Н./

14.08.2008

Пермь 2008

# 1. Функциональное назначение программы, область ее применения, ее ограничения

## 1.1. Назначение программы

Комплекс программ METAS Replication предназначен для решения задачи *тиражирования данных в распределенных динамически настраиваемых информационных системах*, созданных на основе CASE-технологии METAS.

Данные в базах данных (БД) информационных систем (ИС) целесообразно хранить ближе к месту их обработки, то есть на локальных серверах подразделений. Такой способ расположения данных выгоден с точки зрения сокращения времени доступа к ним, уменьшения нагрузки на каналы связи, повышения стабильности системы в целом и информационной безопасности. Базы данных территориально распределенных бизнес-систем (предприятий, организаций, структур управления и т.п.) также становятся *территориально распределенными*, при этом требуется обеспечить оперативный обмен информационными ресурсами между узлами системы, интеграцию отдельных подсистем.

Для решения задачи интеграции следует применять технологии распределенных баз данных (РБД) и тиражирования данных, использовать открытые стандарты, промышленные интеграционные платформы (например, Microsoft BizTalk). Последние ориентированы, в первую очередь, на бизнес-процессы. Использование открытых стандартов значительно упрощает интеграцию систем, но само по себе проблему не решает.

Технологии распределенных баз данных и тиражирования служат не только для организации обмена информацией, документами, но и приближения данных к месту их обработки, что неизбежно приводит к созданию нескольких копий данных в распределенных БД. Таким образом, возникает задача синхронизации БД, на решение которой и направлены технологии распределенных баз данных и тиражирования данных.

Проблема интеграции ИС обостряется особенно в гетерогенной среде, поскольку встроенные механизмы в коммерческих системах управления базами данных (СУБД) не рассчитаны на работу в таких условиях. При этом проблема еще более усложняется при создании динамически адаптируемых ИС, допускающих реструктуризацию данных, т.к. структуры БД в локальных узлах могут различаться, что усложняет процесс реплицирования: требуется тиражировать и синхронизировать не только данные, но и их структуру.

При создании комплекса программ, предназначенных для решения поставленной задачи, были разработаны модели и алгоритмы, соответствующие

двум подходам к решению задачи тиражирования:

- тиражирование данных об объекте;
- тиражирование транзакций (изменений данных, произошедших с момента последней репликации),

а также их программная реализация: на основе предложенных подходов были реализованы два компонента, функционирующих в качестве составной части CASE-системы METAS.

Проведенный в ходе исследовательской работы анализ позволил выделить особенности существующих подходов, которые могут быть использованы для решения поставленной задачи, а также определить присущие этим методам ограничения.

## 1.2. Область применения программы

Данное приложение может быть использовано *для реализации тиражирования данных и схем в ИС*, разработанным на основе CASE-технологии METAS для различных предметных областей.

По причине возрастающей децентрализации информационных ресурсов вопросам их интеграции уделяется все большее внимание. Большинство современных промышленных СУБД имеют встроенные средства реплицирования данных несколькими способами. Однако, если требуется организовать информационный обмен между системами, реализованными на разных платформах, использование таких средств становится невозможным. Поэтому *алгоритмы тиражирования для ИС, создаваемых на основе технологии METAS, допускающей настройку на различные платформы, в общем случае должны быть платформенно-независимы*.

Подходы к тиражированию объектов администрирования в операционных системах зачастую имеют ограничение, связанное с фиксированной структурой этих объектов.

При реализации программных продуктов различных разработчиков ими *были разработаны подходы, позволяющие менять схемы, создавать новые объекты, однако их реализация носит ограниченный характер*, что позволяет их применять только в ограниченных областях.

Одной из основных проблем, стоящих перед разработчиками многих современных ИС, является постоянное изменение требований и условий деятельности учреждений, объединенных использованием единой ИС. Изменения должны оперативно вноситься в функционирующие компоненты по возможности без переписывания программного кода. Поэтому разрабатываемые компоненты тиражирования должны учитывать возможность модификации модели предметной области (БД, пользовательского интерфейса, шаблонов

отчетов и бизнес-процессов, в частности, появление новых операций и пр.). Реализация таких функций обеспечивает *расширяемость, адаптируемость* ИС к новым условиям.

Многие системы тиражирования основаны на синхронной схеме реплицирования, при которой узел-поставщик данных отслеживает результат фиксации отправленных данных на узле-приемнике. Отличительной особенностью предлагаемого подхода к тиражированию является отсутствие требования постоянного оперативного (on-line) соединения между узлами. Это может оказаться необходимым в условиях ненадежных каналов связи и позволяет передавать пакет тиражирования любым удобным способом (на носителе, по электронной почте и т.д.). Возможность тиражирования данных ИС в *асинхронном режиме* – одна из основных задач.

Некоторые исследовательские работы посвящены тиражированию однотипных объектов (например, библиографических записей), которые хоть и допускают изменение их структуры, но не предусматривают наличие связей между ними, в то время как само понятие реляционной модели данных, на которой основано большинство ИС, предполагает наличие отношений между сущностями.

Разрабатываемые в рамках системы METAS средства должны обеспечить возможность *тиражирования взаимосвязанных данных в соответствии со схемами, создаваемыми пользователями* в зависимости от их потребностей в обмене информацией в различных предметных областях.

Реализация компонентов тиражирования данных об объектах ИС в рамках CASE-технологии METAS, обеспечивающей функционирование ИС в режиме интерпретации данных, позволяет оперировать с метаданными системы напрямую, что обеспечивает возможность выбора данных для разработки схемы реплицирования *в терминах предметной области* ИС и делает возможным *автоматическое включение в пакет тиражирования данных о связанных между собой объектах*.

*Степень обоснованности научных положений*. Результаты настоящей работы были получены на основе использования аппарата теории множеств, теории алгоритмов, методов оптимизации, теории системного анализа, а также теории проектирования баз данных и информационных систем. Программная реализация разработанных методик и алгоритмов опирается на технологии объектно-ориентированного анализа и проектирования, структурного и модульного программирования.

### 1.3. Ограничения использования программы

Комплекс программ тиражирования является составной частью CASE-системы METAS. Программное ядро CASE-системы METAS (MDK METAS) позволяет настраиваться на различные программные платформы, работать под управлением различных операционных систем Microsoft, использовать для создания информационных систем различные реляционные СУБД и источники данных, для которых существуют драйверы ODBC.

Однако в данной версии средств тиражирования используются средства создания резервных копий используемых СУБД, в частности Microsoft SQL Server, что ограничивает возможности применения системы.

Ограничение может быть снято путем разработки специальных драйверов для различных СУБД и включения их в разработанный комплекс.

Для функционирования runtime-компонентов необходимо установить .NET Framework (распространяется бесплатно), драйверы ODBC (при установке операционной системы) и СУБД (можно использовать, в частности, Microsoft SQL Server Express, которая распространяется бесплатно).

Возможности масштабирования системы определяются возможностями настройки на различные платформы.

Ограничения использования системы определяются только требованиями лицензионной чистоты и требованиями, предъявляемыми перечисленными выше программными средствами, применяемыми как для разработки ИС, так и для организации ее функционирования.

## 2. Техническое описание программы

При реализации приложения использована многоуровневая модель ИС, создаваемых на основе CASE-технологии METAS. Ниже описана модель, основные принципы функционирования приложения и алгоритмы работы программных компонентов, а также программная реализация средств тиражирования.

### 2.1. Математическая модель информационной системы

CASE-средство METAS работает в интерпретирующем режиме, в основе функционирования системы лежит концепция метаданных, описывающих все объекты, операции и документы информационной системы. Метаданные являются многоуровневыми, и на каждом уровне решается своя задача и предоставляется свой программный интерфейс для взаимодействия с данным уровнем.

*Физический уровень* – это метаданные, описывающие физическое хранение

данных об объектах ИС в БД, схему БД. Служат основой для метаданных логического уровня.

*Логический уровень* – это метаданные, описывающие структуру объектов ИС их взаимосвязи и поведение в терминах предметной области. В основу метаданных этого уровня положена диаграммы классов UML. Логический уровень базируется на физическом в процессе функционирования системы, но при создании ИС разработчик описывает предметную область ИС именно на логическом уровне (предметная область системы, построенной в METAS, описывается на логическом уровне и представляется в виде сущностей и связей между ними), а метаданные физического уровня и соответствующие им структуры данных в БД строятся автоматически с помощью средств реструктуризации CASE-системы.

Метаданные содержат также информацию об отображении данных (*презентационный уровень системы*), включают запросы и шаблоны для формирования отчетов (*модель репортинга*) и т.д. Возможно расширение функциональности системы за счет введения новых операций над сущностями, реализуемое подключением динамических библиотек. Таким же образом к системе могут быть подключены нестандартные элементы управления. Список операций и элементов управления также описывается метаданными ИС.

### *Логический уровень*

В качестве математической модели логического уровня системы можно использовать граф $G_l(V_l, E_l)$, где $V_l = (e_1,...,e_p), p \in N$ – множество сущностей; $E_l = (r_1,...,r_q), q \in N$ – множество связей между ними, $r_i = (e_j, e_k), i = 1..q;\ e_j, e_k \in V_l$.

Направление дуги определяется типом связи между сущностями:

- «1:0..1» – от «1» к «0..1»,
- «1:M» – от «1» ко «M»,
- «0..1:M» – от «0..1» ко «M»,
- «M:M» – дуга двунаправлена,
- «*a..b*:*c..d*» – дуга двунаправлена.

С каждой вершиной графа $e \in V_l$ свяжем множество элементов, обозначающее атрибуты сущности *e*: $Attr(e) = \{a_{e,0}, a_{e,1},..., a_{e,tn}\}$, каждый атрибут состоит из <*name, ref*>, где *name* – имя атрибута; *ref* – если значение атрибута выбирается из справочника, то указывает на соответствующую таблицу-справочник $a_{ref} \in V_{ph}$, в противном случае – пустое значение $a_{ref} = \varnothing$.

### *Физический уровень*

Опишем физический уровень представления данных в системе METAS. Для этого введем граф $G_{ph}(V_{ph}, E_{ph})$, в котором $V_{ph} = \{t_1, t_2, ..., t_{pn}\}, pn \in N$ – множество таблиц в БД; $E_{ph} = \{r_1, r_2, ..., r_{pm}\}, pm \in N$ – множество связей между ними, $r_i = (t_j, t_k), i = 1..pm;\ t_j, t_k \in V_{ph}$.

Направление дуги от $t_j$ (начальной вершины) к $t_k$ (конечной вершине) определяется типом связи между таблицами: «1:M» – от «1» ко «M». Все остальные типы связей в системе моделируются с помощью данного, то есть на физическом уровне в реляционной модели других типов связи нет. Представление связи «многие-ко-многим» организуется через промежуточную таблицу посредством логической модели CASE-технологии METAS (рис. 2.1), следовательно, дуги графа могут быть только однонаправленными.

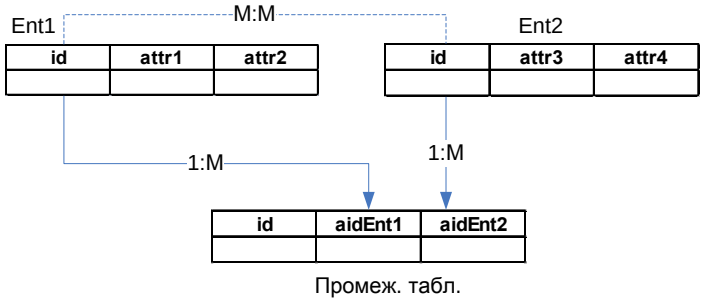


***Рис. 2.1. Представление связи «M:M»***

Свяжем с каждой вершиной графа $t \in V_{ph}$ множество элементов вершины *Fields(t)*, обозначающее поля таблицы *t*: $Fields(t) = \{f_{t,0}, f_{t,1}, ..., f_{t,n_t}\}$, каждый элемент представляет собой четверку значений <*name, type, req, pe*>, где *name* – имя поля таблицы; *type* – представляет тип поля таблицы; *req* – обозначает, является ли наличие значение этого поля обязательным, принимает значения из множества {*true, false*}; *pe* – если поле является ссылкой на первичной ключ другой таблицы, то указывает на нее ( $f_{pe} \in V_{ph}$), иначе имеет пустое значение ( $f_{pe} = \varnothing$).

Будем считать, что элемент $f_{t,0}$ представляет ключевое поле таблицы *t*.

Таким образом, каждая вершина $t \in V_{ph}$ теперь представляет собой структуру таблицы в БД.

Экземпляр данных (строка) *o* таблицы *t* включает множество значений полей $Val(o) = \{v_0, v_1, ..., v_{n_t}\}$; $Val(f, o) \in f_{type}, f \in Fields(t)$ – значение поля $f$ в строке *o*.

Каждое значение $v_i$ строки *o* соответствует полю $f_i$ этой таблицы:

$$Val(f_i, o) = v_i, i = 0..n_t.$$

Будем обозначать значение ключевого поля (идентификатора) строки *o* так:

$$Id(o) = Val(f_0, o) \in N .$$

Зададим местоположение хранения данных (строк таблицы) из множества $Place = \{db, pk, tr\}$, база данных ИС, пакет тиражирования или транзитный пакет. Тогда $O(t, place), t \in V_{ph}, place \in Place$ – множество строк в таблице *t*, *place* задает местоположение объектов, база данных ИС или пакет.

Будем обозначать $O(t, db) = O_{db}(t)$ и $O(t, pk) = O_{pk}(t)$.

Строка однозначно определяется таблицей, значением ключевого поля и местоположением хранения. Введем отображение (функцию), которое ставит им в соответствие экземпляр данных:

$$GetRow(key, t, pl) = o,\ key = Id(o), o \in O(t, pl), t \in V_{ph}, pl \in Place .$$

Зафиксируем произвольную связь между таблицами $r = (t_1, t_2)$. Тогда дочернее поле связи таблицы $t_2$, которое указывает на ключевое поле таблицы $t_1$, определяется следующим образом:

$$CA(r) = (f \in Fields(t_2) : f_{pe} = t_1), r = (t_1, t_2) \in E_{ph}.$$

Для удобства дальнейших рассуждений, введем конструкцию, позволяющую получить значение дочернего поля связи указанного объекта

$$CAV(r, o) = Val(CA(r), o), r = (t_1, t_2) \in E_{ph}, o \in O(t_2, place) .$$

### *Связь между уровнями*

Пусть $MT(e) = t, e \in V_l, t \in V_{ph}$ – главная таблица сущности *e*; $S(e), e \in V_l$ – множество справочников сущности e.

Отношение «M:M» между сущностями реализуется в системе при помощи вспомогательной таблицы. Ей будет соответствовать функция:

$$Mid(r = (e_1, e_2)) = \begin{bmatrix} t_m, (e_2, e_1) \in E_l \\ \varnothing, (e_2, e_1) \notin E_l \end{bmatrix} .$$

Опишем соотношения вершин и дуг графа логической модели и элементов графа физической модели.

Любой вершине (сущности) графа ЛМ соответствует множество таблиц на физическом уровне, состоящее из главной таблицы и таблиц-справочников:

$$(\forall e \in V_l)(\exists T_e \subset V_{ph}) : T_e = MT(e) \cup S(e) .$$

Любой связи «1:M» или «M:1» на логическом уровне соответствует одна связь на физическом.

Связь «M:M» моделируется через промежуточную таблицу, при этом

главные таблицы сущностей состоящих в связи, имеют отношение «1:M» с промежуточной таблицей:

$$(\forall r = (e_1, e_2) \in E_l, e_1, e_2 \in V_l)(\exists E_r \subset E_{ph}):$$
$$E_r = \{(t_1, t_2) \in E_{ph} \mid t_1, t_2 \in V_{ph}, t_1 = MT(e_1), t_2 = MT(e_2), Mid(r) = \varnothing\} \vee$$
$$\vee \{(t_1, t_m) \in E_{ph}, (t_2, t_m) \in E_{ph} \mid t_1, t_2, t_m \in V_{ph}, t_1 = MT(e_1), t_2 = MT(e_2), t_m = Mid(r) \neq \varnothing\}$$

Все множество вершин на физическом уровне можно выразить следующим образом:

$$V_{ph} = \left\{ \bigcup_{e \in V_l} MT(e) \cup S(e) \cup \left( \bigcup_{\substack{d \in V_l, \\ (e,d) \in E_l}} Mid(e,d) \right) \right\}.$$

Это объединение главных таблиц и справочников всех сущностей, а также всевозможных вспомогательных таблиц, реализующих связь «M:M».

Между атрибутами сущности и полями главной таблицы можно установить взаимно однозначное соответствие. Зафиксируем некоторую сущность $e \in V_l$ и ее главную таблицу $t = MT(e) \in V_{ph}$.

Тогда поля таблицы будут представлены множеством

$$Fields(t) = \{f_0, f_1, ..., f_{tn}\},$$

а атрибуты сущности множеством такой же размерности

$$Attr(e) = \{a_0, a_1, ..., a_{tn}\}.$$

Между полями таблицы и атрибутами сущности можно установить взаимно однозначное соответствие. Иначе говоря, существует такое отображение *fa*, что

$$fa: \begin{cases} f_0 \leftrightarrow a_0 \\ \ldots \\ f_{tn} \leftrightarrow a_{tn} \end{cases}, \text{ то есть } fa(f_i) = a_i,\ fa(a_i) = f_i, i = 0..tn.$$

### *Дополнительные обозначения*

Введем следующие обозначения, с помощью которых буду описаны алгоритмы:

- $O(e) = O_{db}(MT(e)), e \in V_l$ – множество экземпляров сущности $e$ из базы данных ИС; $Id(o) \in N, o \in O(e)$ – идентификатор объекта $o$ сущности $e$;
- $CAV(r,o) = (CAV(rt,o) : r = (e_1, e_2), rt = (MT(e_1), MT(e_2))) \in N$, $rt \in E_{ph}, r \in E_l, o \in O(t, place)$ – значение атрибута, представляющего связь «M:1» и являющегося ссылкой на родительскую сущность ($e_1$) объекта $o$;
- $req(e) = \{\forall a \in Attr(e) : f = fa(a), f_{req} = true\}$ – множество обязательных

атрибутов сущности *e*;

- $RM1(e) = \{\forall r = (e_j, e_k) \in E : (e_k = e) \wedge (e, e_j) \notin E\}$ – множество связей «M:1» сущности *e* (все входящие в нее дуги, не имеющие обратных);
- $R1M(e) = \{\forall r = (e_j, e_k) \in E : (e_j = e) \wedge (e_k, e) \notin E\}$ – множество связей «1:M» сущности *e* (все исходящие дуги, не имеющие обратных);
- $RMM(e) = \{\forall r = (e_j, e_k) \in E : (e_j = e) \wedge (e_k, e_j) \in E\}$ – множество связей «M:M» сущности *e* (все двунаправленные дуги).

Введенные обозначения будут использованы в разделе, описывающем подходы к решению задачи тиражирования. Она позволит строго определить понятие схемы тиражирования и более формально подойти к разработке соответствующих алгоритмов. При этом становится возможным выполнять рассуждения и записывать операторы на псевдокоде, основанные на аппарате теории множеств. А это, в свою очередь, позволяет абстрагироваться от некоторых технических моментов реализации, и сосредоточить внимание на сути методов тиражирования. Дополнительным плюсом является простота оценки сложности предложенных решений.

## 2.2. Описание подходов к решению задачи тиражирования в CASE-системе METAS

Реализуется два метода: *тиражирование данных* об объектах ИС и *тиражирование транзакций*.

Создание реплик данных требует также и реплицирования схем данных.

### *Метод тиражирования данных об объекте*

Рассматриваемое средство тиражирования должно позволять создавать копию данных, связанных с некоторым объектом ИС, на одном узле, и извлекать (восстанавливать) данные из этой копии на другом узле. При создании копии пользователь должен выбирать экземпляр сущности (*начальный*, или *корневой*, объект), информацию о котором требуется передать на другие узлы, а также формировать список *дочерних сущностей*, связанных с начальной. Для каждой сущности из этого списка должны быть переданы в пакет только те экземпляры, которые имеют отношение к выбранному объекту. При этом начальная сущность может иметь обязательные связи (ссылки) на *родительские сущности*. Для создания корректной копии при включении сущности в список вместе с ней должны быть добавлены все ее родительские сущности.

При переносе данных из одной системы в другую могут быть выявлены различия в модели данных, поэтому при тиражировании данных необходимо передать информацию и об их структуре.

Список сущностей и начальный объект образуют некоторую *схему тиражирования* данных. Она может быть сохранена на узле-источнике для создания резервной копии данных через некоторое время по той же самой схеме. Сформированная копия включает в себя эту схему, которая сериализуется в XML-файл.

Общая схема создания копии и извлечения из нее данных представлена на рис. 2.2.

На вход подсистемы создания копии данных (Backup) компонента тиражирования поступает БД ИС и схема тиражирования. Основываясь на этой схеме, указанная подсистема выполняет обход таблиц БД, помещая в пакет тиражирования выбранные данные. Помимо этого, пакет содержит и модель данных, которая может отличаться от модели на узле-приемнике, а также начальный объект. Таким образом, в пакет попадает схема тиражирования, поскольку передаваемая модель данных, вообще говоря, включает список сущностей. Сформированный пакет тиражирования любым способом (по локальной сети, посредством электронной почты, на носителе информации и т.п.) доставляется на узел-приемник и поступает на вход подсистемы восстановления копии (Restore).

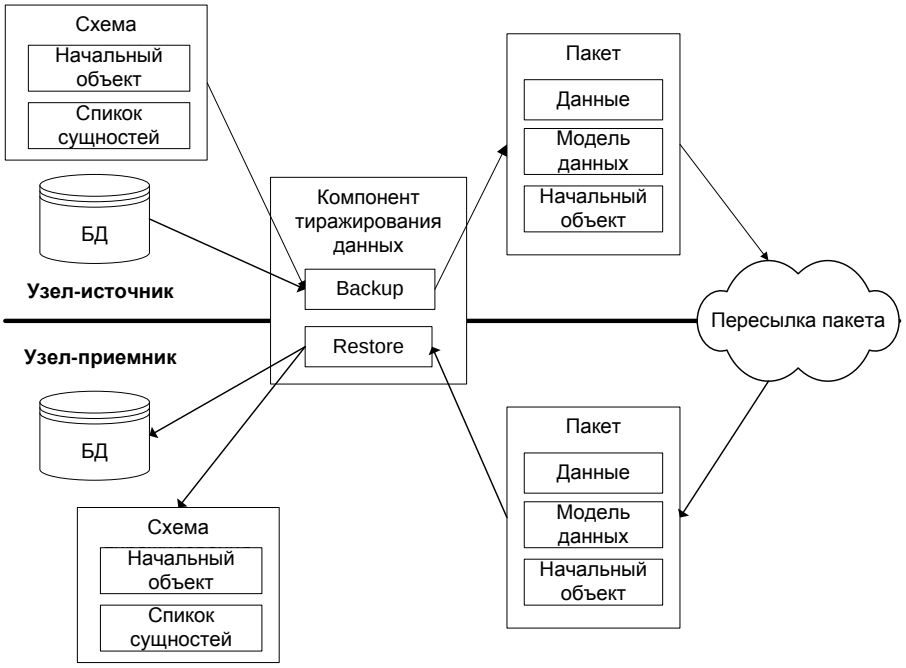


***Рис. 2.2. Процесс создания и восстановления копии***

В первую очередь, данная подсистема осуществляет обновление модели данных (метаданных) на узле-приемнике. Затем происходит извлечение данных в рабочую БД, и в случае успеха, схема тиражирования также может быть сохранена на узле.

Схема работы компонента более подробно обсуждается далее.

### *Понятие схемы тиражирования*

Схема тиражирования состоит из:

– списка сущностей, выбранных пользователем $V_{sc} \subset V_l$;

– списка атрибутов каждой выбранной сущности, необходимых для сопоставления при восстановлении данных $\forall e \in V_{sc} : \exists IdentAt(e) \subset Attr(e)$, причем они включают все обязательные атрибуты сущности плюс, возможно, некоторые другие:

$$IdentAt(e) = req(e) \cup nodefa(e), nodefa(e) \subset Attr(e);$$

для каждого атрибута задается *точность сопоставления*

$$\forall a \in IdentAt(e) : \exists EqAc(a) \subset Accuracy$$

здесь множество *Accuracy* задает всевозможные наборы параметров сравнения значений (полное, с учетом регистра, с учетом пробела и др.);
Факт того, что значения атрибутов (или полей) совпадают с точностью *ea* будем записывать следующим образом:

$$Val(f, o) \overset{ea}{\cong} Val(f, p),\ o \in O(t, pl1), p \in O(t, pl2), f \in Fields(t), t \in V_{sc};$$

– правила сопоставления объектов: требуется равенство значений всех атрибутов или хотя бы одного; правило задается так:

$$EqType(e) \in EqTypes = \{all, one\}, e \in V_{sc};$$

– корневой сущности $e_{start} \in V_{sc}$;

– корневого объекта $o_{start} \in O(e_{start})$.

Модель данных, образованная списком сущностей $V_{sc}$, представляет собой некоторую подсхему логической модели.

*Графом схемы тиражирования* называется подграф графа логической модели $G_l$, образованный редукцией вершин (сущностей), не включенных в схему тиражирования (из множества вершин $V_l$ удаляются все вершины $V_l \setminus V_{sc}$, а из множества дуг – все дуги, инцидентные этим вершинам).

Граф схемы тиражирования определим следующим образом:

$$G_{sc}(V_{sc}, E_{sc}),$$

где $V_{sc} = (e_1, ..., e_n) \subset V_l, n \in N, n \le p$ – множество сущностей, заданных схемой;
$E_{sc} = (r_1, ..., r_m) \subset E_l, m \in N, m \le q$ – множество связей между ними;
$r_i = (e_j, e_k), i = 1..m;\ e_j, e_k \in V_{sc}$.

Исходя из определения графа схемы тиражирования, множество $E_{sc}$

строится так:

$$E_{sc} = E_l \setminus \{r = (e_j, e_k) \in E_l \mid e_j \in V_l \setminus V_{sc} \vee e_k \in V_l \setminus V_{sc}\},$$

т.е. из множества дуг логической модели удаляются все дуги, инцидентные сущностям, не включенным в схему тиражирования.

### *Алгоритм создания резервной копии*

*Операция создания резервной копии* по указанной схеме означает построение графа $G_{pk}(V_{pk}, E_{pk})$, где $V_{pk} \subset V_{ph}, E_{pk} \subset E_{ph}$.

Учитывая то, что схема задает набор сущностей $V_{sc}$, необходимых для копирования, становится возможным определить множество таблиц в этом графе

$$V_{pk} = \left\{ \bigcup_{e \in V_{sc}} MT(e) \cup S(e) \cup \left( \bigcup_{\substack{d \in V_{sc}, \\ (e,d) \in E_{sc}}} Mid(e,d) \right) \right\}.$$

При этом в пакет тиражирования должны попасть объекты, имеющие отношение к начальному. Это означает построение множества $O(t, pk) \subset O(t, db)$ для каждой таблицы $t \in V_{pk}$.

Указанный алгоритм выполняет рекурсивный обход графа схемы тиражирования в глубину. В процессе обхода последовательно просматриваются связи текущей вершины с родительскими сущностями «M:1», затем «1:M» и «M:M». Для каждой связи вызывается алгоритм *TransIdAndCopyXX*, который выполняет преобразование множества идентификаторов объектов данной сущности в множество идентификаторов сущности, состоящей в отношении с данной. При этом происходит копирование объектов сущности в пакет тиражирования. При внесении в пакет очередного экземпляра, функция проверяет, не был ли он туда внесен ранее, если да – то он повторно не вносится, и не возвращается в множестве *NewIds*. Это обеспечивает конечность алгоритма (количество объектов в БД конечно). Если множество *NewIds* не является пустым, значит, для экземпляров с такими идентификаторами нужно выполнить обход – рекурсивно вызвать следующий алгоритм:

// На входе: $e \in E_{sc}$ – очередная сущность, посещаемая алгоритмом обхода, связи
// которой требуется просмотреть;
// $Ids \subset N$ – множество идентификаторов экземпляров этой сущности;
// ExcludeR –связь, по которой был осуществлен переход в e; по ней возвращаться
// не следует
*proc DoTraversalByScheme(* $e \in V_{sc}, Ids \subset N, ExcludeR \in E_{sc}$ *)*
*begin*
    // обход связей *M:1*

```
    for each r ∈ RM1_e do
        if r ≠ ExcludeR then
        begin
            NewIds ← Ids;
            // Преобразование идентификаторов и копирование объектов
            // r_1 – первая компонента вектора r, сущность со стороны 1
            TransIdAndCopyM1(e, r_1, r, ref NewIds);
            if NewIds ≠ ∅ then DoTraversalByScheme(r_1, NewIds, r);
        end;
    // обход связей 1:M
    for each r ∈ R1M_e do
        if r ≠ ExcludeR then
        begin
            NewIds ← Ids;
            TransIdAndCopy1M(e, r2, r, ref NewIds);
            if NewIds ≠ ∅ then DoTraversalByScheme(r2, NewIds, r);
        end;
    // обход связей M:M
    for each r ∈ RMM_e do
if r ≠ ExcludeR then
    begin
        NewIds ← Ids;
        TransIdAndCopyMM(e, r2, r, ref NewIds);
        if NewIds ≠ ∅ then DoTraversalByScheme(r2, NewIds, r);
    end;
end;
```

Перед запуском алгоритма необходимо скопировать корневой объект в пакет тиражирования и в качестве входных параметров задать корневую сущность $e_{start}$ и идентификатор корневого объекта $Id(o_{start})$: $DoTraversalByScheme(e_{start}, Id(o_{start}), \varnothing)$.

Пусть $N$ – число вершин в графе, $K$ – общее число строк в таблицах. Одна и та же вершина может быть посещена несколько раз, причем в самом худшем случае K раз (для каждого объекта подъем по родительской связи). Таким образом, рекурсия может быть выполнена $O(K \cdot N)$. Позже будет показано, что функции *TransIdAndCopyXX* имеют сложность $O(N + K^2)$. Общая сложность алгоритма $O(K^3 N + K N^2)$.

**Переход по связи «1:M»**

При переходе к дочерней вершине анализируется принадлежность объектов ко всем ближайшим предкам, зависящим от стартовой вершины. Независимо от того, из какой родительской вершины мы пришли в данную, нужно выбирать все

экземпляры сущности, у которых родительские строки (зависимые от корневой) уже были ранее помещены в пакет. Это обеспечит выбор только тех объектов, которые относятся к начальному.

Множество вершин, зависимых от начальной, должно расширяться по мере обхода. Действительно, вначале зависимыми вершинами являются все вершины дерева стока по связям «1:M», то есть такие вершины, до которых из корневой существует путь в графе, состоящий только из направленных дуг. Например, на рис. 2.3 для стартовой вершины $e_1$ зависимыми являются вершины $e_2$ и $e_3$. Затем алгоритм обхода перейдет в вершину $e_4$, которая является корнем другого поддерева. Вообще говоря, вершина $e_5$ теперь должна являться зависимой от $e_4$.

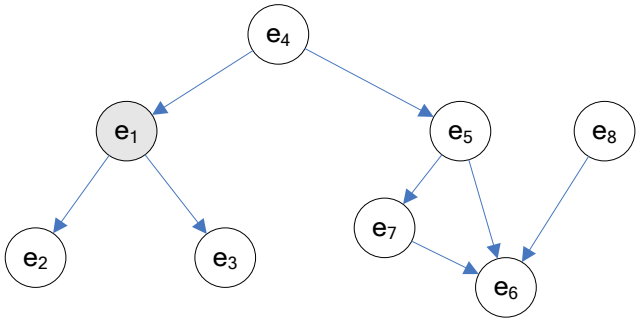


***Рис. 2.3. Граф схемы, содержащей только связи «1:M»***

На протяжении работы алгоритма будем строить множество зависимых вершин

$$DepEnts \subset V_{sc}.$$

Вершины, входящими в это множество, будем называть *окрашенными*, а операцию добавления вершины в это множество – *окраской* вершины.

Рекурсивный алгоритм окрашивания дерева стока дочерних вершин выглядит следующим образом:

```
// На вход передается вершина-сущность e ⊂ Vsc
proc MarkupDependentEntity( e ⊂ Vsc )
begin
// красим эту вершину
   DepEnts ← DepEnts ∪ {e};
  for each r ∈ R1M_e do
       if (r is not 0..1:M) and r2 ∉ DepEnts then
            MarkupDependentEntity( r2 );
end;
```

Сложность алгоритма *O(N)*, где *N* – число вершин графа.

Рассмотрим алгоритм преобразования множества идентификаторов и

копирования соответствующих экземпляров сущностей в пакет при переходе по связи «1:M». Вначале идет разветвление в зависимости от типа связи. Если вершина окрашена, значит мы идем по строгой связи «1:M». Тогда составляется множество всех связей с окрашенными родительскими сущностями и вызывается алгоритм *CopyWhichParentExists* копирования всех зависимых строк данной сущности, которая возвращает множество идентификаторов, ранее не присутствовавших в пакете:

// на входе: *e* – сущность, объекты которой надо скопировать в пакет;
// *RP* – множество связей с родительскими сущностями.
// Объект будет скопирован, если в пакете уже содержатся те сущности,
// на которые он ссылается
*proc CopyWhichParentExists(* $e \subset V_{sc}, RP \subset E_{sc}$ *)*
*begin*
// все такие объекты главной таблицы, у которых все значения дочерних
// атрибутов содержатся как идентификаторы в соответствующих
// родительских таблицах

$$ForCopy \leftarrow \left\{ \forall o \in O_{db}(MT(e)) : \bigwedge_{rp \in RP} CAV(rp, o) \in Id(O_{pk}(MT(rp_1))) \right\};$$

// возвращает идентификаторы только вновь добавляемых объектов

$$NewIds \leftarrow Id(ForCopy \setminus O_{pk}(MT(e)));$$

// добавление в пакет

$$O_{pk}(MT(e)) \leftarrow O_{pk}(MT(e)) \cup ForCopy;$$

*return NewIds;*
*end;*

Сложность алгоритма определяется построением множества *ForCopy*, $O(K^2)$, где *K* – число строк в таблицах.

Если вершина не окрашена, значит мы переходим по связи «0..1:M». Алгоритм выбирает из главной таблицы сущности идентификаторы тех строк, у которых атрибут-ссылка на родителя принадлежит текущему множеству *Ids*. Затем происходит копирование выбранных строк в пакет. Алгоритм *CopyTable* также возвращает множество идентификаторов, помещенных в пакет и ранее в нем не существовавших.

// *t* – таблица, *Objs* – множество объектов для копирования
*proc CopyTable(* $t \subset V_{pk}, Objs \subset O(t, place)$ *)*
*begin*
// возвращает идентификаторы только вновь добавляемых объектов

$$NewIds \leftarrow Id(Objs \setminus O_{pk}(t));$$

// добавление в пакет

$O_{pk}(t) \leftarrow O_{pk}(t) \cup Objs;$

*return NewIds;*

*end;*

Сложность алгоритма – квадратичная относительна числа строк в таблице. Приведем алгоритм *TransIdAndCopy1M*, суммируя все вышесказанное.

// На вход поступает сущность $e_1$, с которой осуществляется переход,
// сущность $e_2$, соответствующая стороне «M»,
// множество идентификаторов *Ids*, которое необходимо преобразовать
*proc TransIdAndCopy1M(* $e_1 \in V_l, e_2 \in V_l$, *ref* $Ids \subset N$ *)*

*begin*

*if* $e_2 \in DepEnts$ *then*

*begin*

// составим множество связей с окрашенными родителями

$RP \leftarrow \{\forall r \in RM1(e_2) : r_1 \in DepEnts\};$

$Ids \leftarrow CopyWhichParentExists(e_2, RP);$

*end*

*else*

*begin*

$ForCopy \leftarrow \{\forall o \in O_{db}(MT(e_2)) : CAV((e_1, e_2), o) \in Ids\};$

$Ids \leftarrow CopyTable\ (O_{db}(MT(e_2)), ForCopy);$

*if* $Ids \neq \varnothing$ *then MarkupDependentEntity(*$e_2$*);*

*end;*

*end;*

Обе ветки условия имеют квадратичную сложность относительно числа строк в таблице и линейную относительно числа вершин графа, что и определяет общую сложность алгоритма $O(N + K^2)$.

**Переход по связи «M:1»**

Переход по связи «M:1» имеет смысл в том случае, если родительская сущность не является окрашенной. Чтобы получить новое множество идентификаторов, необходимо организовать просмотр всех строк указанной сущности и для каждой из них выделить ссылку на предка.

// $e_1$ – сущность, с которой осуществляется переход,
// сущность $e_2$, соответствующую стороне «1»
// множество идентификаторов *Ids*, которое необходимо преобразовать
*proc TransIdAndCopyM1(* $e_1 \in V_l, e_2 \in V_l$, *ref* $Ids \subset N$ *)*

*begin*

```
if e2 ∉ DepEnts then
begin
    ForCopy ← ⋃_{o∈O_db(MT(e1))} CAV((e1, e2), o) : Id(o) ∈ Ids ;
    Ids ← CopyTable(e2, ForCopy) ;
    if Ids ≠ ∅ then MarkupDependentEntity(e2);
end
else Ids = ∅ ;
end;
```

Сложность алгоритма – $O(N + K^2)$.

**Переход по связи «M:M»**

При переходе алгоритма по связи «M:M» возможны следующие ситуации:

Целевая вершина (та, в которую пытаемся перейти) имеет хотя бы одного окрашенного потомка, при этом возможно, что не прямого, имеющего свои экземпляры в пакете. Это означает, что обе сущности, вообще говоря, являются зависимыми от корневой, и новых объектов по связи получено быть не должно.

Через связь происходит переход в другой подграф, то есть дуга связи является мостом в графе.

Первая ситуация предусматривает случай, представленный в виде примера на рис. 2.4.

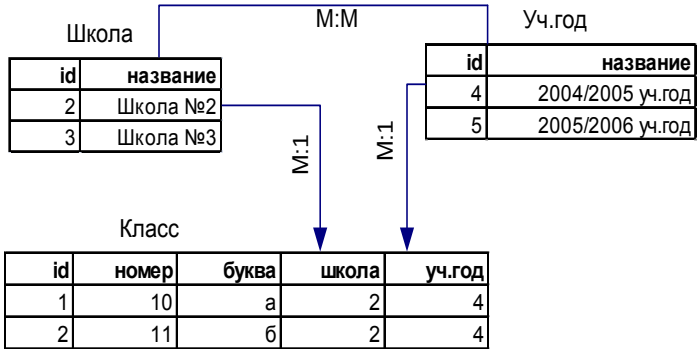


***Рис. 2.4. Пример, показывающий проблему при переходе по связи «M:M»***

Предположим, что в качестве корневого объекта выбран 10 «а» класс, соответствующая строка которого будет скопирована в пакет перед началом рекурсии. Затем алгоритм выберет, например, экземпляр сущности «Уч.год» с *id*=4. Далее, рассматривая связь «M:M», можно получить все школы, которые существовали в 2004/2005 учебном году. Однако это конечно же является неверным решением, поскольку нас интересует только «Школа №2», в которой находится 10 «а» класс. В данной ситуации переход по связи «M:M» вообще не должен быть осуществлен, поскольку экземпляры и сущности «Уч.год», и

сущности «Школа» должны быть получены лишь при переходе по связям «M:1». Тем не менее, нельзя забывать о поддержке промежуточной таблицы. Указанный случай можно распознать, проверяя, не имеет ли целевая сущность дочерних окрашенных сущностей. При этом дочерняя сущность должна иметь какие-либо экземпляры в пакете тиражирования.

Функция *HasPaintedChildren* организует рекурсивный просмотр дочерних сущностей и возвращает истину, если найдена окрашенная вершина с ненулевым числом экземпляров в пакете.

Обработка второй ситуации происходит тривиально – необходимо через промежуточную таблицу определить список идентификаторов целевой вершины и перенести соответствующие строки в пакет тиражирования.

```
// На вход поступает сущность e1, с которой осуществляется переход,
// e2 – целевая сущность,
// множество идентификаторов Ids, которое необходимо преобразовать
proc TransIdAndCopyMM( e1 ∈ Vl , e2 ∈ Vl , ref Ids ⊂ N )
begin
  if HasPaintedChildren(e2) & ( e2 ∉ DepEnts )
  // целевая вершина имеет хотя бы одного окрашенного потомка (ситуация 1)
    begin
      // Новые объекты получены не будут
      Ids ← ∅;
      // Копируем промежуточную таблицу связи
      t ← Mid((e1, e2)); t1 ← MT(e1); t2 ← MT(e2);
      CopyWhichParentExists(t, {(t1, t), (t2, t)});
    end
  else
  // через промежуточную таблицу определить список идентификаторов
  // целевой вершины и перенести соответствующие строки
  // в пакет тиражирования (ситуация 2)
    begin
      // Вначале копируются строки промежуточной таблицы, у которых
      // атрибут связи с исходной сущностью e1 принадлежит указанному
      // множеству Ids
      t ← Mid((e1, e2)); t1 ← MT(e1); t2 ← MT(e2);
      ForCopy ← {∀o ∈ O(t) : CAV((t1, t), o) ∈ Ids};
      Opk(t) ← Opk(t) ∪ ForCopy;
      // Преобразование множества идентификаторов – среди
      // скопированных только что строк, уникально выбираются ключи
      // целевой сущности
```

$Ids \leftarrow \bigcup_{o \in ForCopy} CAV((t_2, t), o)$;

// Скопировать строки целевой сущности

$CopyTable(e_2, \{(\forall o \in e_2) : Id(o) \in Ids\})$;

*if* $Ids \neq \varnothing$ *and* $e_2 \notin DepEnts$ *then MarkupDependentEntity(*$e_2$*);*

*end; // of if*

*end; // of proc*

Все рассмотренные ситуации имеют квадратичную сложность относительно числа строк в таблице, определение множества окрашенных родителей – линейно относительно *N*, что и определяет общую сложность алгоритма $O(N + K^2)$.

**Журнализация работы алгоритма создания копии**

Журнал представляет собой таблицу, каждая строка которого содержит имя физической таблицы, сущность и некоторую пометку к ней:

$$< table, ent, label > \in LogTable,\ table \in V_{sc}, ent \in V_l .$$

Порядок строк важен, добавление строки возможно только в конец журнала. Журнал пополняется каждый раз, когда в пакет была произведена вставка строк – добавляется информация о том, куда была произведена вставка (в какую таблицу или сущность), и почему эта вставка произошла (пометка). Метки выбираются из множества

$$LogRowTypes = \{Usual, GoUp, RefBook\}.$$

*GoUp* означает, что при копировании был произведен переход к родительской сущности, *RefBook* – что был скопирован справочник.

Алгоритмы переходов по связям необходимо дополнить, добавив команды для ведения журнала.

При переходе по связи «1:M» после операторов добавления в пакет нужно произвести проверку, были ли реально вставлены строки и если да, то сделать запись в журнале:

*If* $Ids \neq \varnothing$ *then AddLogRow(*$e_2$*, Usual);*

Для связи «M:1» нужно установить пометку, что был выполнен переход наверх:

*If* $Ids \neq \varnothing$ *then AddLogRow(*$e_1$*, GoUp);*

Отношение «M:M» мы рассматривали с точки зрения двух возможных ситуаций. При восстановлении данных нужно вначале обрабатывать таблицы сущностей, состоящие в связи, и только потом вносить строки промежуточной таблицы. Первая ситуация предполагает вначале внесение строк целевой таблицы, затем промежуточной, поэтому в журнале строки могут быть помечены

как *Usual*. Во втором случае вставка строк осуществляется только в промежуточную таблицу. В последнем случае вначале копируются данные промежуточной таблицы (на физическом уровне это переход по связи «1:M»), а затем целевой сущности (подъем по родительской связи). Поэтому пометки соответственно должны быть *Usual* и *GoUp.*

### *Алгоритм восстановления данных*

*Операция восстановления данных* из резервной копии подразумевает обновление данных, содержащихся в БД ИС. Пакет тиражирования содержит множество таблиц $V_{pk} \subset V_{ph}$, заполненных строками *O(t, pk).* Необходимо организовать просмотр всех строк, при анализе очередной строки в пакете нужно искать запись в локальной БД, которая представляет тот же самый объект. В случае успешного поиска требуется обновить несовпадающие атрибуты, иначе добавить объект в множество *O(t, db).*

Правила сопоставления для каждой сущности задаются схемой тиражирования, по которой была создана резервная копия. Строки считаются *похожими*, если у них совпадают значения идентифицирующих атрибутов *IdentAt(e)* с точностью *EqAc(e)*, где $e \in V_l$. При распознавании строк могут возникать неопределенности, связанные с неполнотой или ошибочностью данных. Разрешение этих неточностей целесообразно возложить на плечи администратора. Поэтому, если для конкретного объекта в пакете не был найден соответствующий объект в БД или их было найдено несколько, то такой экземпляр данных будет помещен в *промежуточный* (или *транзитный*) пакет вместе с результатами распознавания. Для того чтобы сохранить целостность данных, *необходимо помещать и все его дочерние объекты* (которые на него ссылаются).

Промежуточный пакет *TempPk* состоит из множества таблиц $V_{tr} \subseteq V_{pk}$, каждая строка которой состоит из элементов вида <*o, ols, ol*>, где $o \in O_{tr}(t), t \in V_{pk}$ – экземпляр данных в таблице; $ols \subset O_{db}(t)$ – множество результатов распознавания, строки из локальной БД; $ol \in O_{db}(t) \cup \{\varnothing\}$ – строка, с которой должно быть установлено соответствие, или пустой элемент, если должна быть добавлена новая строка. Данная компонента отражает действия администратора над пакетом.

Поскольку ключевое поле объекта должно быть уникальным, алгоритм должен учитывать возможность изменения ключа объекта при его вставке в БД ИС.

Будем считать, что граф $G_{pk}(V_{pk}, E_{pk})$ и соответствующий ему граф

логической модели $G_l(V_l, E_l)$ на узле-приемнике изоморфны графам на узле-источнике. Иначе говоря, подсхема модели данных, образованная схемой тиражирования, одинакова на узлах.

На вход алгоритму подается копия данных – фрагмент БД узла-источника и лог-таблица. Вначале необходимо просмотреть эту таблицу и составить порядок обхода таблиц в пакете. Справочники являются самостоятельными таблицами, которые не могут иметь родительских связей, поэтому все они могут быть обработаны первыми. Строки помеченные как *Usual*, добавляются в конец формируемого порядка обхода. Последовательности пометок *GoUp* в журнал означают, что осуществляется подъем по родительским связям и поэтому, они должны быть обработаны в обратном порядке, чтобы сущности более верхнего уровня оказались перед дочерними сущностями.

Обход таблиц из $V_{sc}$ представлен алгоритмом ниже. В зависимости от типа обрабатываемой таблицы, вызывается подпрограмма ее вставки с различными параметрами.

```
for i=1 to |Order| do
    begin
        lt ← Order_i, lt ∈ LogTable;
        table ← lt_table, table ∈ V_sc;
        label ← lt_label, label ∈ LogRowType;
        if lt_ent ≠ ∅ then
            // таблица представляет сущность
            InsertTable(table, IdentAt(lt_ent));
        else
            if label=RefBook then
                // таблица является справочником
                InsertTable(table, {F1 | F=Fields(table)});
            else //таблица является промежуточной для связи M:M
                InsertTable(table, {F1, F2 | F=Fields(table)});
    end;
```

*InsertTable* просматривает все строки указанной таблицы в пакете и выполняет замещение данных в БД ИС. При этом гарантируется, что если была произведена вставка строки с другим ключевым значением, то в пакете на нее будут обновлены все ссылки.

Второй параметр *InsertTable* задает множество полей, по которым необходимо проводить сопоставление строк пакета и локальной БД. Все справочники в системе на физическом уровне устроены одинаково – они содержат ключевое поле и значение. Поэтому сопоставление происходит по

значению. При восстановлении вспомогательной таблицы для связи «M:M», идентифицирующими полями являются ссылки на родительские таблицы. Действительно, порядок обхода таблиц гарантирует, что родительские таблицы будут обработаны перед вспомогательной. Поэтому, она будет гарантировано содержать ключевые поля объектов из локальной БД.

Рассмотрим алгоритм вставки таблицы.

// *t* – таблица, *IF* – идентифицирующие поля
*proc InsertTable(* $t \in V_{sc}, IF \subset Fields(t)$ *)*
*begin*
// стереть признак изменения ключа
*ClearIdChanged(t);*
// перебор всех строк в таблице *t* из пакета
*for each* $o \in O_{pk}(t)$ *do*
*begin*
// поиск объекта в локальной БД по атрибутам *IF*
*if EqType(t) = all then*

$$ols \leftarrow \{d \in O_{db}(t) : \underset{f \in IF}{\wedge} Val(f,d) \overset{ea}{\cong} Val(f,o),\ ea = EqAc(f)\}$$

*else* $ols \leftarrow \{d \in O_{db}(t) : \underset{f \in IF}{\vee} Val(f,d) \overset{ea}{\cong} Val(f,o),\ ea = EqAc(f)\}$*;*

*if* $|ols| = 1$ *and UnRec(o)=false then*
*begin*
// строка найдена, при том единственная
$ol \leftarrow ols_0$*;*
*F* ← *Fields(t);*
// заменить все различающиеся поля, кроме ключа
*for each* $f \in F, f \neq f_0$ *do*
*if* $Val(f,ol) \neq Val(f,o)$ *then Val(f, ol)* ← *Val(f, o);*
// если в локальной БД другой ключ, то заменяем ключ в пакете
*if* $Id(o) \neq Id(ol)$ *then UpdateRefs(t, Id(o), Id(ol));*
*end*
*else begin*
// поместить строку в промежуточный пакет
$TempPk(t) \leftarrow TempPk(t) \cup \{< o, ols, \varnothing >\}$*;*
// пометить все дочерние строки – они тоже должны
// попасть в промежуточный пакет
*MarkUnrec(t, Id(o));*
*end;*
*end;*
*end;*

Сложность одной итерации цикла $O(K \cdot A)$, где $K$ – общее число строк в таблицах, $A$ – число атрибутов. Поскольку цикл перебирает все строки таблицы, общая сложность процедуры $O(K^2 \cdot A)$.

Алгоритм восстановления данных вызывает указанную процедуру для каждой таблицы, поэтому общее число итераций цикла *for each* будет составлять $K$ раз. Таким образом, общая сложность алгоритма $O(K^2 \cdot A)$.

После того, как администратор вручную установил соответствие объекта из промежуточного пакета с объектом системы (задал *ol* в пакете), или принял решение о вставке нового экземпляра ($ol = \varnothing$), должен сработать следующий алгоритм:

```
// <o, ols, ol> ∈ TempPk(t) – строка промежуточного пакета;
// t ∈ V_pk – таблица, строка которой попала в промежуточный пакет;
// o ∈ O(t, tr) – объект из промежуточного пакета;
// ol ∈ O_db(t) ∪ {∅} – соответствие с объектом системы
if ol ≠ ∅ then
   begin
      F ← Fields(t);
      // заменить все различающиеся поля, кроме ключа
      for each f ∈ F, f ≠ f_0 do
          if Val(f, ol) ≠ Val(f, o) then Val(f, ol) ← Val(f, o);
      // если в локальной БД другой ключ, то заменяем ключ в пакете
      if Id(o) ≠ Id(ol) then UpdateRefs(t, Id(o), Id(ol));
   end
else
   begin
      // нужно вставить новую строку
      // попробовать вставить с тем же ключом, поиск совпадений
      olk ← {d ∈ O_db(t) : Id(d) = Id(o)};
      if olk ≠ ∅ then
          begin // ключ занят другим объектом, найдем уникальный ключ
              NewId ← {n ∈ N : ∧_{d ∈ O_db(t)} Id(d) ≠ n};
              // обновить все ссылки на этот ключ
              UpdateRefs(t, Id(o), NewId);
              Id(o) ← NewId;
          end;
      // вставить объект в БД
      O_db(t) ← O_db(t) ∪ {o};
   end;
```

Сложность приведенного алгоритма $O(K \cdot A)$.

Алгоритм *UpdateRefs* должен просматривать все дочерние связи таблицы и заменять старые значения ключа на новые. Изменение ссылки у конкретного объекта нужно фиксировать, поскольку все объекты таблицы обрабатываются последовательно. Нельзя допускать, чтобы изменение ссылки происходило два раза в рамках обработки одной таблицы.

```
// t – таблица, OldId – старый ключ, NewId – новый ключ
proc UpdateRefs( t ∈ V_sc , OldId ∈ N , NewId ∈ N )
 begin
    // просмотр всех дочерних связей
    for each r = (t, tc), tc ∈ V_sc do
        // просмотр дочерних объектов в пакете, имеющих старый ключ
        for each o ∈ O_pk(tc) : CAV(r, o) = OldId ∧ IdChanged(o) = false do
        begin
            CAV(r, o) ← NewId;
            IdChanged(o) ← true;
        end;
 end;
```

Очевидно, алгоритм имеет квадратичную сложность относительно числа строк в таблице, поскольку он содержит вложенный цикл. Однако ранее было принято искать сложность относительно общего числа строк во всех таблицах, обозначенное числом *K*. В таком случае вычислительная сложность будет *O(K)*.

Поскольку объекты могут иметь несколько ссылок на родительские, перед обработкой очередной таблицы требуется очищать признак изменения ссылки у всех дочерних строк этой таблицы. Эту операцию реализует следующий алгоритм.

```
// t – таблица
proc ClearIdChanged( t ∈ V_sc )
begin
  // просмотр всех дочерних связей
  for each r = (t, tc), tc ∈ V_sc do
     // просмотр всех дочерних объектов в пакете
     for each o ∈ O_pk(tc) do IdChanged(o) ← false;
end;
```

Алгоритм имеет сложность *O(K)* , где *K* – общее число строк в таблицах.

Подпрограмма *MarkUnrec* должна пометить все дочерние строки как нераспознанные с той целью, чтобы они также попали в промежуточный пакет.

Это позволит сохранить целостность данных в системе.

// *t* – таблица, *Id* –ключ объекта
*proc MarkUnrec(* $t \in V_{sc}, Id \in N$ *)*
*begin*
// просмотр всех дочерних связей
*for each r = (t, tc),* $tc \in V_{sc}$ *do*
// просмотр дочерних объектов в пакете, имеющих ссылку на ключ
*for each* $o \in O_{pk}(tc) : CAV(r, o) = Id$ *do*
*UnRec(o)* ← *true;*
*end;*

Алгоритм имеет сложность *O(K)* , где *K* – общее число строк в таблицах.

### *Тиражирование транзакций*

Основной особенностью данного подхода к тиражированию является способ определения множества данных, необходимых для пересылки. Предположим, что некоторые два узла системы имеют одинаковую базу данных. На узле-источнике пользователи вносят изменения в БД, которые фиксируются специальной подсистемой. Очевидно, что если на узле-приемнике повторить всю последовательность манипуляций с данными, то базы данных узлов снова придут в одинаковое состояние.

При генерации пакета используется журнал изменений (транзакций) данных, журнал изменения метаданных и журнал репликаций. Процесс генерации состоит в последовательной обработке журналов изменений и создании для каждой записи журнала соответствующей записи в пакете. Получение записи, с которой необходимо начать обработку журналов изменений происходит из журнала репликаций, который содержит дату последней репликации в приемник.

Процесс интерпретации пакета на стороне приемника состоит из двух этапов:

– Интерпретация метаданных (если они присутствуют в пакете);
– Интерпретация данных.

Оба этапа подразумевают последовательную обработку соответствующих разделов пакета. Вначале устанавливается соответствие сущностей из раздела пакета метаданных с метаданными базы-приемника.

При безошибочной идентификации и успешном внесении изменений в базу метаданных, происходит интерпретация данных. Если внесение метаданных в базу невозможно, или администратор отказался принимать метаданные из пакета, то компонент репликации может настроить компонент миграции с целью

внесения данных в БД. Таким образом, появляется возможность импорта данных без требования идентичности структур баз данных источника и приемника.

Схема, иллюстрирующая данный подход, представлена на рис. 2.5.

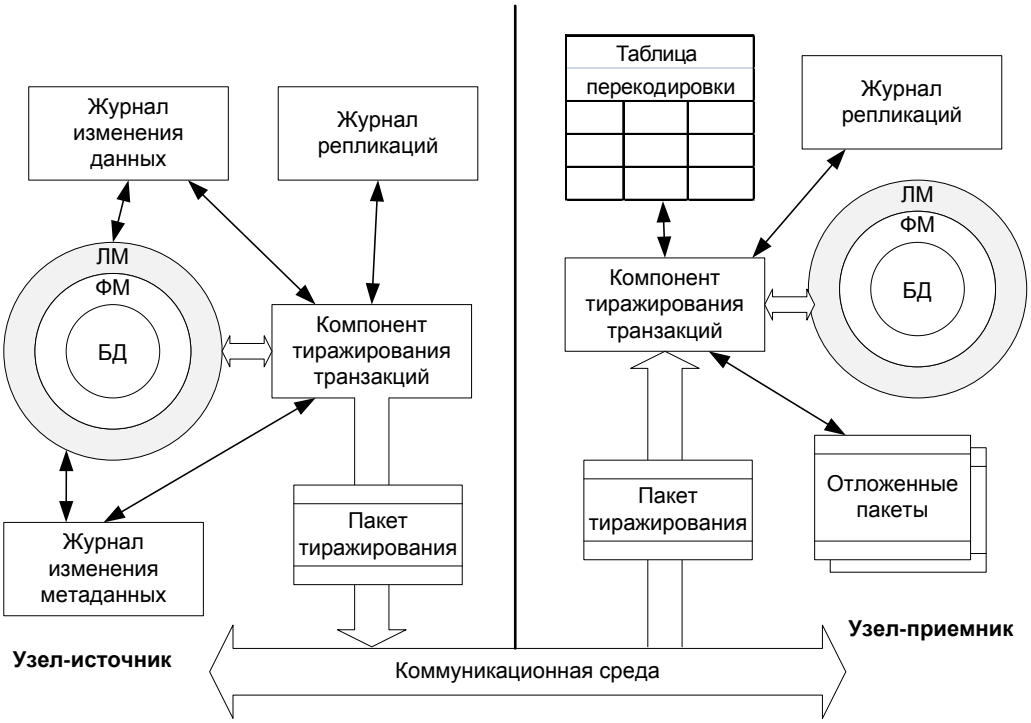


***Рис. 2.5. Обобщенная схема тиражирования транзакций***

Тиражирование транзакций предполагает журнализацию производимых операций над объектами в системе. *Журнал изменения данных* представляет собой множество элементов *Jour*, каждый из которых состоит из четверки <*e, ob, op, t*>, где $e \in V_l$ – сущность, $ob \in O(e)$ – объект, $op \in Oper = \{add, upd, del\}$ – произведенная операция, $t \in N$ – метка времени.

Схема тиражирования, как и в случае тиражирования копий, состоит из множества сущностей, идентифицирующих атрибутов и правил их сопоставления.

Пакет тиражирования будет состоять из некоторой выборки элементов журнала. Необходимо помещать в него изменения, произошедшие с момента последней репликации на конкретный узел-приемник. Рассмотрим алгоритм создания пакета.

// *dt* – метка времени, соответствующая моменту последнего тиражирования
*proc GenPack(* $dt \in N$ *)*
*begin* // выбор всех записей, начиная с некоторой метки времени,
// и относящихся к входящей в схему сущности
$PK \leftarrow \{\forall jr \in Jour,\ jr =< e, ob, op, t >: e \in V_{sc}, dt > t\}$;
*end;*

Сложность алгоритма – линейная относительно числа записей в журнале.

При включении записи в пакет, естественно должны быть включены все значения атрибутов объекта.

При распознавании объекта могут возникать неопределенности, связанные с неполнотой или ошибочностью данных. Разрешение этих неточностей целесообразно возложить на плечи администратора. Поэтому при идентификации пакета тиражирования вначале создается *промежуточный* (или *транзитный*) пакет, в который включаются все записи с результатами распознавания. Важно понимать, что порядок внесения изменений в объекты БД должен совпадать с порядком записей в журнале.

Каждый элемент транзитного пакета представляет собой шестерку $<e, ob, op, t, ols, ol>$, где $e \in V_l$, $ob \in O_{pk}(e)$, $op \in Oper$, $t \in N$, $ols \subset O_{db}(e)$ – множество распознанных объектов, $ol \in O_{db}(e) \cup \{\varnothing\}$ – объект, с которым должно быть установлено соответствие, или пустой элемент, если объект должен быть добавлен. Данная компонента отражает действия администратора над пакетом.

Рассмотрим алгоритм первоначальной идентификации пакета тиражирования транзакций. Он состоит в последовательном просмотре записей, идентификации каждой и помещении результатов в промежуточный пакет. При этом может быть изменена операция, которую требуется осуществить (например, операция *add* заменяется на *upd*, если в БД приемника найден объект).

*proc IdentPack()*
*begin*
// перебор всех записей пакета
*for each* $pr \in PK$ *do*
*begin*
$\neg pr \leftarrow < e, ob, op, t >$, $e \in V_l$, $ob \in O_{pk}(e)$, $op \in Oper$, $t \in N$,
$ols \subset O_{db}(e)$, $ol \in O_{db}(e) \cup \{\varnothing\}$;
// поиск объекта в локальной БД по заданным атрибутам
$IF \leftarrow IdentAt(e);$
*if EqType(t) = all*
*then* $ols \leftarrow \{d \in O_{db}(t) : \underset{f \in IF}{\wedge} Val(f,d) \overset{ea}{\cong} Val(f,ob), ea = EqAc(f)\}$
*else* $ols \leftarrow \{d \in O_{db}(t) : \underset{f \in IF}{\vee} Val(f,d) \overset{ea}{\cong} Val(f,ob), ea = EqAc(f)\}$;
*if* $|ols|=0$
*then* // ни один объект не сопоставлен
*if (op = add)*
*then* $TransPk \leftarrow TransPk \cup < e, ob, add, t, ols, \varnothing >$

```
        else if (op = upd)
            then  TransPk ← TransPk∪ < e,ob,add,t,ols,∅ >
            else if (op = del)
                then  TransPk ← TransPk∪ < e,ob,del,t,ols,∅ >
                else // найден один или более объектов
                    if (op = add)
                     then  TransPk ← TransPk∪ < e,ob,upd,t,ols,∅ >
                    else if (op = upd)
                        then  TransPk ← TransPk∪ < e,ob,upd,t,ols,∅ >
                       else if (op = del)
                            then  TransPk ← TransPk∪ < e,ob,del,t,ols,∅ >
    end;
end;
```

Если принять, что число записей в пакете пропорционально числу $K$ – числу строк в таблицах, то сложность алгоритма будет $O(K^2 \cdot A)$.

После того, как администратор вручную установил соответствие всех объектов из промежуточного пакета с объектами системы и принял решение о вставке новых экземпляров, необходимо осуществить еще один просмотр записей, теперь уже в промежуточном пакете. Алгоритм обработки каждой записи аналогичен алгоритму, рассмотренному для задачи восстановлении резервной копии.

## 2.3. Структура и реализация программного продукта

Программный продукт включает несколько компонентов, выделенных по функциональному принципу, реализующих перечисленные выше подходы к тиражированию. Их реализация описана выше

### *Компонент тиражирования данных об объекте*

Данный компонент тиражирования можно рассматривать в некотором приближении как средство тиражирования снимка БД, из таблиц которой сделана горизонтальная вырезка. Объем передаваемых данных может быть очень большим. При передаче возникает необходимость строго соблюдать структуру и формат передаваемых данных. Преобразование форматов следует свести к минимуму, чтобы, во-первых, не допустить искажения самих данных, а во-вторых, обеспечить высокую производительность процесса тиражирования.

Таким образом, целесообразно организовать взаимодействие компонента с данными напрямую, то есть на физическом уровне ИС. Другим вариантом мог бы являться логический уровень, но работа на нем с коллекциями объектов

(множеством строк) представляется неэффективной.

Компонент тиражирования данных об объекте функционально можно разбить на такие составляющие:

- подсистема создания копии данных (резервной копии);
- подсистема восстановления копии;
- управление схемами тиражирования;
- сопряжение с ядром CASE-системы.

Схему взаимодействия этих подсистем друг с другом и с другими составляющими METAS отражает рис. 2.6.

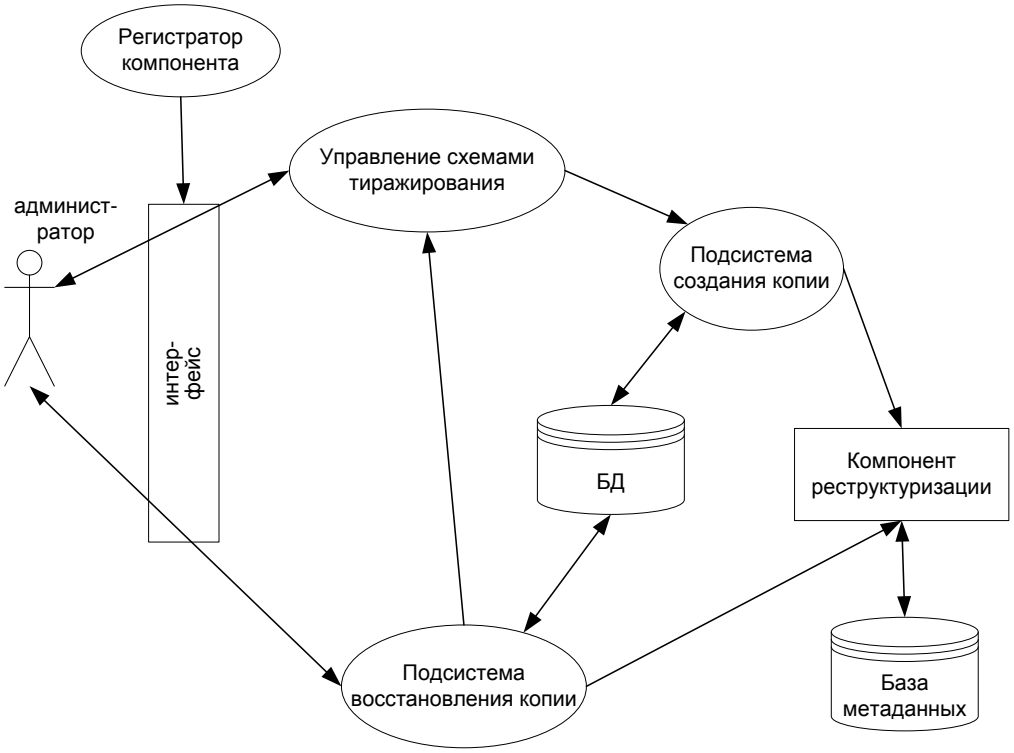


***Рис. 2.6. Схема взаимодействия подсистем компонента***

Подсистема сопряжения с ядром выполняет первоначальную загрузку компонента, добавляет соответствующие пункты меню в интерфейс ИС и настраивает взаимодействие с базой данных и метаданных.

### Структура данных копии

Необходимо определиться со структурой формируемой копии. Здесь есть несколько вариантов:

- типизированный файл;
- открытый XML-формат;
- формат файла БД или ее резервной копии.

Преимущество использования собственного бинарного формата (типизированного файла) состоит в его независимости от исходной и целевой

СУБД. Специально разработанная структура обеспечит компактность. Однако просмотр такого файла становится невозможен другими средствами, на его структуру будет настроен только рассматриваемый компонент тиражирования, что не удовлетворяет условию открытости. В случае появления новых типов данных, не запланированных заранее, может возникнуть необходимость изменения всей структуры.

Эту проблему решает использование языка разметки XML, который допускает добавление новых тэгов, при этом устаревшие версии компонента тиражирования будут продолжать работать. Минусом такой универсальности, простоты и открытости оказывается громоздкость. Тиражирование снимка БД во многих случаях означает большой объем передаваемых данных. Использовать DOM модель в таких условиях не следует, но даже и потоковое чтение XML (SAX модель) вызовет дополнительные накладные расходы на его разбор.

Формат файла БД оказывается наиболее удобным в процессе формирования копии. Достаточно создать новую БД, включив в нее необходимые таблицы из рабочей БД ИС с аналогичной структурой, и произвести копирование некоторых строк. При создании копии нередко требуется знать, какие из объектов уже были помещены в пакет. Эту задачу успешно можно переложить на плечи СУБД, что очевидно ведет к простоте реализации. Основной же недостаток такого формата – привязка к конкретной СУБД. Файл базы данных, созданной, например, в Microsoft SQL Server, не удастся импортировать в Oracle без использования специальных утилит.

Тем не менее, на данном этапе разработки компонента, используется именно последний вариант. Данные для тиражирования размещаются во временной БД под управлением Microsoft SQL Server. После формирования этой базы, происходит создание ее резервной копии средствами указанной СУБД. На узле-приемнике эта копия разворачивается также во временную БД, из которой происходит копирование в рабочую базу ИС. Такой выбор обусловлен, помимо всего прочего, простотой отладки. Чтобы просмотреть содержимое пакета тиражирования, можно легко воспользоваться соответствующими средствами SQL Server. Кроме того, компонент тиражирования создавался на этапе проектирования ИС «Образование Пермской области», стоимость внедрения которой можно было уменьшить за счет использования бесплатной версии Microsoft SQL Server Express для небольших узлов системы.

В будущем, представляется разумным использовать некий комбинированный вариант. В силу простоты реализации, копию данных можно формировать во временной БД, а затем, для обеспечения переносимости на другие СУБД, упаковывать ее в типизированный файл или представлять в виде XML. То есть заменить технологию формирования резервной копии БД своей собственной.

**Подсистема создания копии данных**

Данная подсистема включает в себя два класса:

- *Backup* – содержит основную логику работы программы создания копии. Реализует функции, соответствующие алгоритмам *DoTraversalByScheme* и *TransIdAndCopyXX*.
- *Saver* – реализация физического создания копии, может рассматриваться как СУБД-зависимый модуль. Осуществляет алгоритмы *CopyWhichParentExists*, *CopyTable*, журнализацию работы процесса создания копии.

Для представления множества идентификаторов в указанных алгоритмах используются динамические списки.

Непосредственно перед началом обхода необходимо произвести сохранение модели данных ИС, соответствующей списку сущностей, участвующих в процессе тиражирования. Этот список образует некоторый подграф общего графа системы и является некоторой подсхемой. Сохранение осуществляется подсистемой тиражирования компонента реструктуризации модели данных. Данную операцию реализует класс *RestructSerializer*, методу которого передается описание подсхемы в виде сущностей и связей между ними. Сериализация производится в отдельный XML-файл.

Рассмотрим алгоритм *CopyTable*, который производит копирование записей с указанными значениями ключевого поля из рабочей БД в пакет. Чтобы обеспечить конечность алгоритма создания копии, *CopyTable* должен возвращать только те идентификаторы записей, которые ранее в пакете не содержались. Это также необходимо и для исключения повторной вставки строк в таблицы временной БД. Таким образом, для организации этой операции вычитания, нужно знать ключевые поля уже добавленных в пакет строк конкретной таблицы.

Их можно определять одним из двух способов:

- вести справочник идентификаторов добавленных объектов каждой сущности на протяжении всей работы алгоритма;
- определять это множество идентификаторов каждый раз при необходимости.

Очевидно, первый способ более привлекателен в силу своего быстродействия. Тем не менее он может оказаться не пригодным для больших БД, поскольку требует немалых затрат оперативной памяти. В тоже время, может показаться, что определение множества идентификаторов при каждом вызове *CopyTable* серьезно повлияет на производительность. Тем более, что число вызовов этой подпрограммы будет большим. Однако операторы безусловного выбора типа

*SELECT id FROM table*

выполняются СУБД очень быстро. Весьма часто будет происходить такой запрос повторно, на неизменившейся таблице. В таком случае успешно сработают механизмы кэширования, присутствующие во многих современных СУБД. Поэтому предпочтение отдано именно этому варианту.

Для оптимальной реализации операции вычитания множеств, в запросе на выборку существующих идентификаторов присутствует сортировка

*SELECT id FROM table ORDER BY id*

Тогда в процессе перебора всех объектов для копирования, определить, содержится ли он уже в пакете, можно за время меньшее *O(n)*. Действительно, следует организовать двоичный поиск, сложность которого *O(log n).*

Алгоритм *CopyWhichParentExists* копирует все объекты указанной сущности, у которых родительские объекты уже присутствуют в пакете. Это можно сделать, используя рассмотренный выше способ, определив для каждой родительской сущности набор ключевых значений и найдя их пересечение. Однако тогда вопрос производительности встает наиболее остро. Более проще оказывается переложить эту задачу на СУБД, которая оптимальным образом реализует операцию сопоставления полей.

Функция должна формировать и исполнять следующий запрос, приведенный на диалекте языка SQL

*select * from MT(e) WHERE*
*exists(select * from PacketDb.*$rp_1^{(1)}$*.id = BaseDb.CA(*$rp^{(1)}$*)) and …*
*…and exists(select * from PacketDb.*$rp_1^{(n)}$*.id = BaseDb.CA(*$rp^{(n)}$*))*

где *PacketDb* – временная БД, в которую помещается пакет тиражирования; *rp* – элемент множества *RP*, верхний индекс обозначает номер элемента в множестве, нижний – номер компоненты вектора связи; *BaseDb* – основная БД ИС.

Формально, здесь мы видим необходимость выборки с сопоставлением полей из различных баз данных. Такая возможность не входит в стандарт SQL-92 [19] и потому доступна не во всех СУБД. В частности, одним запросом в Microsoft SQL Server такую выборку сделать нельзя. Во избежание этого ограничения предлагается формировать пакет тиражирования прямо в рабочей БД ИС, назначая его таблицам специальные имена, например, добавляя некоторый префикс.

По окончанию работы алгоритма создания копии, содержимое этих таблиц следует перенести во временную БД, после чего их следует удалить. Затем нужно сформировать резервную копию этой БД средствами СУБД. Теперь можно удалить и эту временную базу, в результате чего система вернется в исходное состояние, которое было до выполнения всего алгоритма.

Рассмотрим преобразование множества идентификаторов при переходе по связи «M:1», которое осуществляется алгоритмом TransIdAndCopyM1. Объекты

для копирования выбираются по формуле

$$ForCopy = \bigcup_{o \in O_{db}(MT(e_1))} CAV((e_1, e_2), o) : Id(o) \in Ids .$$

Наиболее очевидной реализацией этой формулы – уникальная выборка значений заданного поля строки таблицы, ключ которой входит в множество *Ids*. На языке SQL это выглядит так

*select distinct CA(e1,e2) from MT(e1) where id in (Ids)*

где идентификаторы из *Ids* указываются через запятую: *in* (*1, 5, 7, …*).

Примечательным оказался тот факт, что когда множество *Ids* достаточно велико (1 000 записей и более), выполнение такого запроса оказывается чрезвычайно медленным на Microsoft SQL Server. Переписывание условной части в виде

*where (id=1) or (id=5) or (id=7) or…*

не привело к положительному результату. К тому же, текст таких запросов может оказаться очень длинным, что таит дополнительную опасность, так как некоторые СУБД ограничивают максимальную длину запроса.

Ощутимо эффективнее работают много коротких запросов, условная часть которых содержит по одному значению идентификатора. При этом исключение повторных значений выборки осуществляется самим компонентом.

В процессе создания копии данных нередко происходит копирование во временные таблицы большого количества строк. Известно, что выполнение подряд одиночных операций вставки происходит неэффективно. По всей видимости, узким местом становится провайдер доступа к данным. Повысить производительность позволяет использование пакетных команд (batch-queries) [18]. Такие запросы содержат в себе несколько простых запросов, разделенных между собой точкой с запятой. К сожалению, не все СУБД их поддерживают, но когда их использовать возможно, то это следует делать.

С этой целью компонент содержит класс *CacheInsert*, позволяющий кэшировать запросы на вставку данных, исходящие от функций *CopyTable* и *CopyWhichParentExists*. В случае накопления максимального установленного количества команд, объект указанного класса осуществляет их объединение и выполняет полученный пакетный запрос. Безусловно, допускается очищение кэша, то есть принудительное исполнение по запросу, посредством посылки соответствующего сообщения объекту *CacheInsert*.

**Подсистема восстановления копии**

Восстановление копии начинается с разбора структуры модели данных. Для этого подсистеме тиражирования компонента реструктуризации передается XML-описание подсхемы. Этот шаг обеспечивает необходимое условие для

начала работы алгоритма восстановления данных – идентичность структуры сущностей и связей между ними на узле-источнике и узле-приемнике.

Перед началом обхода таблиц для вставки необходимо развернуть резервную копию базы, соответствующей пакету тиражирования, во временную БД. Это осуществляется средствами СУБД.

Заострим внимание на некоторых особенностях реализации алгоритма *InsertTable*. Эта подпрограмма осуществляет последовательный перебор строк заданной таблицы в пакете тиражирования и для каждой из них пытается найти соответствующий объект в БД приемника. Сопоставление происходит по заданным полям для сущности, для справочника – по полю значения. Информация о том, по каким полям следует осуществлять поиск поступает на вход алгоритма. Поиск организуется посредством следующей SQL-команды на выборку

*select * from t where IF1 = Val(IF1,o) and …*
*…and IFn = Val(IFn,o)*

где $o \in O_{pk}(t)$ – очередной объект из пакета тиражирования, *IF* – множество идентифицирующих полей, размерностью *n*.

В случае необходимости сопоставление можно провести по одному из заданных атрибутов (например, совпадение ОКПО или ИНН). Тогда в запросе нужно указать не конъюнкцию, а дизъюнкцию, то есть вместо *and* поставить *or*. Понятие точности сопоставления удобно реализовать посредством оператора *LIKE* языка SQL вместо точного равенства.

Если объект в результате такого запроса найден не был, значит либо произошли изменения в одном из его идентифицирующих атрибутов, либо на одном из узлов содержались неточные или ошибочные данные, либо такой объект на узле-приемнике попросту не существовал. В последней ситуации такую сущность необходимо добавить в основную БД, взятую из копии. При этом непременно возникает проблема первичных ключей. Ранее было принято, что ключевые поля в METAS формируются обычным образом, что не обеспечивает их уникальность на различных узлах ИС. Поскольку вставка строки с ключом, который уже имеется в таблице недопустима и блокируется даже на уровне СУБД, требуется изменить один из ключей. Реляционная модель данных предполагает также и связывание таблиц по ключевому полю, поэтому при изменении ключевого поля должны модифицироваться и все ссылки на него. То есть речь идет о каскадном обновлении.

Изменение ключа необходимо производить также и в случае того, когда соответствие между объектами установлено. Не исключено, что дочерние сущности будут сопоставляться по ссылочному атрибуту. Алгоритм

восстановления данных организует такой порядок обхода, при котором гарантируется, что вначале будут обработаны родительские сущности, затем на один уровень вниз – дочерние, и так далее по иерархии.

Обновлять ключевые атрибуты можно или в основной БД узла системы, или во временной БД, приводя ее к непротиворечивому виду с рабочей базой. Разумнее, конечно, второй вариант, потому что, во-первых, мы гарантированно не испортим существующие данные, во-вторых, объем каскадных обновлений будет меньше, поскольку пакет тиражирования содержит некоторое подмножество дочерних сущностей. А при изменении ключа в основной БД придется модифицировать абсолютно все ссылки.

Каскадное обновление реализует алгоритм *UpdateRefs*, выполняющий просмотр дочерних сущностей заданного объекта. Список дочерних сущностей можно получить через ядро METAS, обеспечивающее доступ к метаданным. Изменение ключевого атрибута удобно осуществить SQL-командой

*update tc set CA(r)=NewId where CA(r)=OldId*

где *tc* – таблица дочерней сущности, *CA(r)* – атрибут связи, *NewId* – новый идентификатор, *OldId* – старый идентификатор.

Тем не менее, такой способ использовать недопустимо. При последовательной обработке записей некоторой таблицы может получится так, что первичный ключ изменит свое значение на такое, которое уже содержится в таблице и принадлежит другому объекту. Вследствие чего временная БД придет в неверное состояние, поскольку ссылки будут нарушены. Поясним сказанное на примере (рис. 2.7).

При выполнении распознавания объектов из таблицы «Человек», будет установлено соответствие между строками с фамилией Иванов. В результате потребуется изменение ключевого поля, со значения 601 в 508, в том числе и в таблице «Документ», поскольку идентифицирующими атрибутами для нее являются поля «Номер» и «Человек» (рис. 2.8). Затем алгоритм подойдет к строкам с фамилией Петров, и, аналогичным образом, произведет изменение ПК, с 508 на 706. Рассмотренная выше команда обновления *update* при этом выполнит операцию не над одним документом 789524, принадлежащему Петрову, а затронет обе строки (рис. 2.9).

В итоге связи между таблицами будут нарушены, в нашем примере окажется, что Иванову теперь принадлежит оба документа.

Указанную проблему можно решить, введя запрет на повторное изменение атрибута связи. Таким образом, встает необходимость вести список дочерних объектов, ссылки которых были модифицированы (за это отвечает множество *IdChanged* в алгоритмах предыдущей главы). Более того, нужно как то оповещать об этом СУБД при выполнении запроса на обновление.

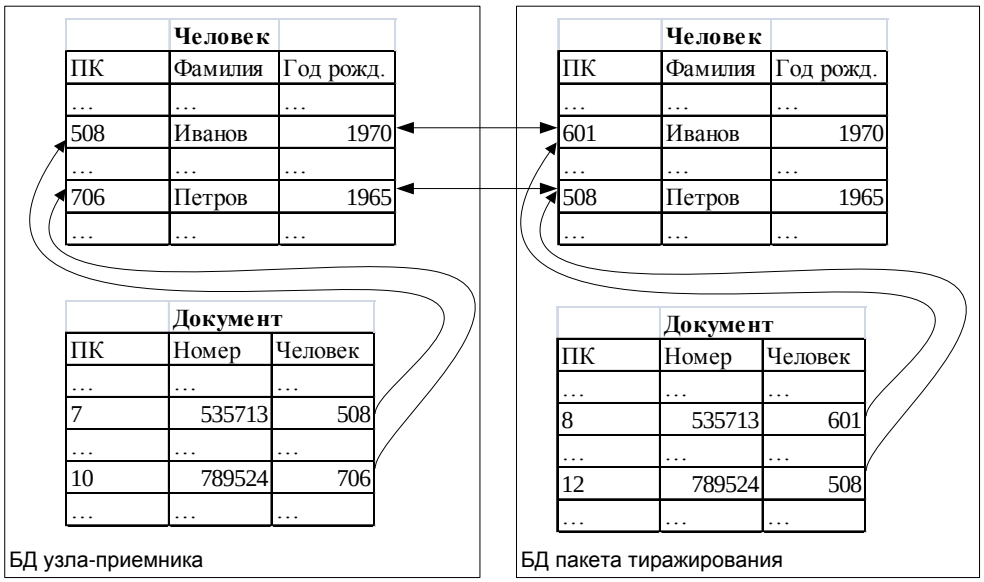


***Рис. 2.7. Пример, отражающий проблему каскадного обновления ПК***

**Документ**

| ПК | Номер | Человек |
|---|---|---|
| … | … | … |
| 8 | 535713 | ~~601~~ 508 |
| … | … | … |
| 12 | 789524 | 508 |
| … | … | … |

***Рис. 2.8. Таблица «Документ» в пакете тиражирования***

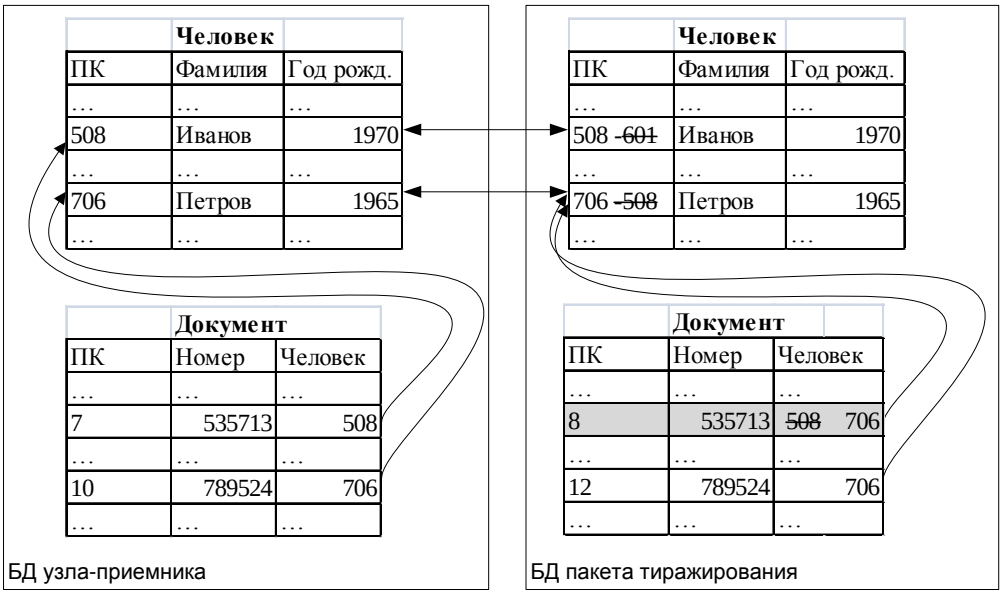


***Рис. 2.9. Ошибочная ссылка***

Выходом является добавление к каждой таблице столбца, логического типа, который показывает, было ли произведено изменение атрибута связи. При этом достаточно лишь одного столбца, поскольку требуется обеспечить

одноразовость изменения только в процессе просмотра одной таблицы. Соответственно, при выборе очередной таблицы указанный флаг должен быть снят.

Таким образом, после распаковки резервной копии, каждая таблица дополняется служебным столбцом *IdChanged*. Запрос на обновление будет выглядеть так

```
update tc set CA(r)=NewId, IdChanged=1
where CA(r)=OldId and IdChanged=0
```

### Управление схемами тиражирования

За просмотр, изменение и запуск процесса тиражирования отвечают следующие классы:

- *SchemeManagerForm* – позволяет просмотреть существующие на узле схемы тиражирования, добавить, удалить или переименовать их. Предоставляет также возможность начать процесс формирования копии.
- *ReplSchemeEditor* – форма для редактирования схем тиражирования.
- *Scheme* – класс, представляющий непосредственно схему, осуществляет объектный доступ к ее свойствам, отвечает за ее физическое сохранение и загрузку.
- *SchemeDescription* – организует сериализацию схемы с целью ее транспортировки.

Форма *ReplSchemeEditor* позволяет создать схему тиражирования, основой которой является список сущностей. Как уже было отмечено, при добавлении в список некоторой сущности необходимо помещать в него и всех ее родителей, чтобы копия получилась корректной. Такой способ формирования списка реализуется на уровне интерфейса, через специальный элемент управления.

Схемы тиражирования хранятся в базе метаданных ИС в специальных таблицах. Соответствующая модель данных показана на рис. 2.10.

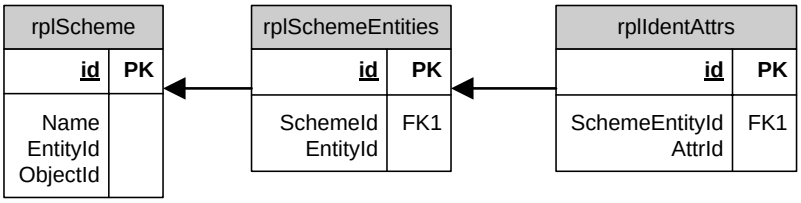


***Рис. 2.10. Модель данных для хранения схем тиражирования***

Таблица *rplScheme* содержит названия схемы, стартовый (корневой) объект и идентификатор его сущности. Поле корневого объекта может быть не определено, это будет означать, что его надо запросить у пользователя перед началом формирования копии, и такая схема является общей для тиражирования сущности (например, тиражирование данных о школе – задается информация,

которая должна попасть в пакет; она является общей для всех школ).

Поскольку схема содержит целый список сущностей, для их хранения требуется дополнительная таблица, *rplSchemeEntities*, связанная отношением «M:1» с *rplScheme*. И для каждой такой сущности определяется список идентифицирующих ее атрибутов в таблице *rplIdentAttrs*.

При формировании копии в пакет тиражирования помещается также и сама схема тиражирования, которая сериализуется в отдельный XML-файл с помощью класса *SchemeDescription*. По окончанию всего процесса восстановления данных, в случае успеха, она сохраняется или обновляется на узле-приемнике. Это может оказаться весьма удобным для обратной пересылки данных через некоторое время по той же самой схеме.

### *Компонент тиражирования транзакций MDK Replication*

Компонент MDK Replication состоит из трех взаимосвязанных программных модулей: Replication, RepLib и RepPresent.

#### Компонент Replication

Модуль *Replication* отвечает за интеграцию с комплексом METAS. Он реализует специальный интерфейс *IComponent*, который содержит метод *Load*, вызываемый при загрузке комплекса. В методе *Load* происходит:

1. Регистрация и добавление пунктов меню на основную форму системы, при помощи которых администратор может работать с компонентом.
2. Подключение к логической модели данных, получение из ядра системы ODBC-строки подключения к БД Logs, соединение с этой базой данных.
3. Подписка на события добавления, изменения, удаления экземпляров сущностей предметной области ИС.
4. Подписка на события добавления, изменения, удаления сущностей предметной области ИС.
5. Подписка на событие начала новой транзакции в системе.
6. Реализация событий добавления, изменения и удаления сущностей и их экземпляров, и их регистрация в журналах изменения метаданных и данных соответственно.
7. Присвоение каждой возникающей транзакции уникального номера.

#### Библиотека базовых классов RepLib

Модуль RepLib реализует основную логику работы алгоритма и включает в себя следующие классы:

- *OperationTypes* – обозначает тип операций в журнале изменений, вставка, удаление или модификация объекта.
- *JournalRecord* – класс, представляющий запись журнала изменений.

Содержит поля:

- *int Entity* – идентификатор сущности;
- *int Object* – идентификатор объекта;
- *OperationTypes Operation* – тип операции, вставка, удаление или изменение;
- *DateTime Date* – дата вносимых изменений;
- *int TransactionId* – уникальный номер транзакции.
- *int OrganizationId* – код организации, в которой зафиксировано изменение.

Фактически данный класс полностью отражает структуру журнала изменений.

– *ChangeJournal* – интерфейс доступа к журналу изменений.
– *RepPackage* – интерфейс доступа к пакету репликации. Выполняет основную работу компонента – генерацию пакета, метаданных, а также их интерпретацию.
– *Organization* – интерфейс к списку организаций, участвующих в репликации.
– *RepTypes* – интерфейс доступа к схемам тиражирования.

Класс *ChangeJournal* служит для обеспечения доступа к журналу изменении. Он осуществляет регистрацию происходящих транзакций, получая соответствующие сообщения от модуля Replication. Журналы хранятся в базе метаданных ИС в специальной таблице. Алгоритм формирования пакета тиражирования также обращается к данному классу для получения изменений, произошедших с момента последней репликации на указанный узел. Записи журнала возвращаются последовательно (в соответствие с концепцией итератора) в хронологическом порядке за исключением тех, которые были получены с узла-приемника при обратной репликации.

Класс *RepPackage* выполняет самые главные функции – генерацию пакета тиражирования и его распознавание. При формировании пакета происходит последовательная обработка вначале записей журнала метаданных, затем – журнала данных, и помещение их в пакет.

Распознавание пакета тиражирования начинается с интерпретации раздела метаданных и их обновления на узле-приемнике. Только в случае успешного завершения этой операции можно переходить к обновлению данных. Однако в ряде случаев автоматическое обновление метаданных без вмешательства администратора может оказаться весьма затруднительным. Например, в структуру сущности был добавлен атрибут, заполнение которого является обязательным. И в то же время для него не задано значение по умолчанию.

Случай, вообще говоря, нетривиальный и искусственный, тем не менее компонент предлагает способ разрешения такой ситуации. Создается временная таблица, в которую помещаются все данные данной сущности и вызывается компонент миграции данных, с помощью которого администратор может вручную установить необходимые соответствия между атрибутами сущности и полями этой таблицы.

Схемы тиражирования представлены классом *RepTypes*. Он содержит методы создания, редактирования, удаления и переименовывания схем, характеризующихся списком сущностей, информацию о которых требуется передавать, и прочими параметрами.

Класс Organization служит для доступа к списку подразделений организации. Для корректной работы механизма обратной репликации требуется, чтобы каждый узел тиражирования имел уникальный номер. Этот код учитывается в журналах изменения и показывает, где была зафиксирована запись. Класс Organization позволяет получить код данного (локального) подразделения. Также он предоставляет доступ к общему списку организаций, информацию которым требуется передавать. Для каждой из них задается схема тиражирования, определяемая классом *RepTypes*.

### Презентационная составляющая компонента репликации – RepPresent

Модуль *RepPresent* содержит следующие формы, организующие пользовательский интерфейс для управления компонентом тиражирования:

- *ChangeJournalForm* – класс, отвечающий за отображение журнала изменений;
- *GenForm* – форма генерации пакета репликации;
- *InterpForm* – форма интерпретирования пакета;
- *HandInterpForm* – форма ручной идентификации данных;
- *OptionsRepForm* – форма настройки компонента.

Форма *ChangeJournalForm* осуществляет представление журнала изменений данных в удобной для администратора форме, позволяя применять несложный фильтр и делать выборку записей по дате.

Форма генерации пакета *GenForm* позволяет запустить алгоритм формирования пакета тиражирования. Для этого задается организация-приемник и расположение файла, который будут сгенерирован. Опционально можно задать последнюю дату репликации. По умолчанию она определяется автоматически из журнала произведенных репликаций.

Форма автоматической интерпретации пакета *InterpForm* служит для разбора пакета тиражирования и запуска алгоритма распознавания. Не

идентифицированные записи помещаются в отложенный пакет, работа с которым реализуется формой *HandInterpForm* ручного установления соответствия между объектами системы и пакета тиражирования. При выборе очередного объекта выводится соответствующий список экземпляров сущностей в рабочей БД.

Форма настройки компонента *OptionsRepForm* позволяет определять и редактировать схемы тиражирования, просматривать список организаций и выбирать для каждой из них схему тиражирования.

**Вложенные транзакции в METAS**

Практически в любой ИС необходимо наличие транзакций. Поэтому CASE-технология METAS должна соответствующим образом осуществлять их поддержку.

Рассмотрим пример, показывающий важность транзакций в информационных системах, созданных на основе METAS. При редактировании некоторого объекта системы (например, «Человек») на одной форме может располагаться несколько объектов разных сущностей («Документ человека»). В частности, если при добавлении сведений о человеке в БД по некоторой причине произошел сбой, то было бы неверным занесение в БД сведений о его документе. Значит, сохранение изменений нескольких объектов из формы следует выполнять в одной транзакции.

Более того, возможна ситуация, когда из одной формы вызывается другая. В итоге получаем, что одна транзакция вложена в другую. В общем случае, степень вложенности не должна быть ограничена.

В настоящее время, большинство современных СУБД поддерживают транзакции и самостоятельно реализуют поддержку свойств ACID. Тем не менее, лишь немногие из них допускают вложенные транзакции. Требование переносимости к разрабатываемой CASE-технологии в частности означает возможность использования различных СУБД. Поэтому вложенные транзакции реализуются посредством ядра METAS следующим образом:

- Все транзакции системы можно рассматривать как дерево.
- Пусть некоторый компонент системы инициирует транзакцию. Если дерево транзакций пусто, то оно создается, и в корень помещается поступившая транзакция. Затем под дерево инициализируется реальная СУБД-транзакция.
- Если дерево не пусто, то добавление в него соответствующей вершины-транзакции. На самом деле, используется та единственная СУБД-транзакция, которая была создана для корня.

– В случае отката в одной из вершин, происходит общий откат всего дерева.
– В случае успешного закрытия всех транзакций дерева, закрывается и СУБД-транзакция.

В некотором смысле, действующий алгоритм работы компонента репликации можно рассматривать, как запоминание действий пользователя на узле-источнике и воспроизведение этих действий на узле-приемнике. В связи с этим, очевидно, необходимо запоминать происходившие при этом транзакции, и учитывать их при обновлении БД-приемника.

Для реализации учета транзакций при тиражировании данных необходимо решить следующие проблемы: отслеживание транзакций, их помещение в пакет репликации и, соответственно, изменения алгоритма интерпретации пакета.

Необходимо отслеживать происходящие в системе транзакции и фиксировать их. Поскольку транзакции касаются изменения данных, то предпочтительнее всего их записывать в журнал изменений данных, так как это облегчит процедуру генерации пакета.

Заметим, что нам не обязательно знать обо всех виртуальных (управляемых ядром METAS) транзакциях. Фактически, это не дает никаких преимуществ, а также осложняет ситуацию хранения дерева в рамках реляционной модели данных. Достаточно реагировать на событие начала СУБД-транзакции (начало всеобъемлющей транзакции). Таким образом, будем иметь последовательность простых (не вложенных друг в друга) транзакций, каждой из которых можно назначить некоторый номер. Это номер должен быть уникален среди всех записей журнала изменений и, по сути, он является искусственным ключом произошедшей транзакции. Его следует указывать при регистрации событий добавления, удаления и изменения объектов ИС.

При последовательной обработке записей журнала изменений данных, требуется заносить для каждой записи уникальный номер соответствующей ей транзакции.

Поскольку записи в пакете репликации следуют в хронологическом порядке (отсортированы по времени возникновения событий по возрастанию), то можно утверждать, что соответствующие им транзакции также будут расположены последовательно. После успешной интерпретации пакета, при внесении данных в БД ИС, требуется закрывать предыдущую транзакцию и создавать новую, если значение ее ключевого поля сменилось в процессе обхода журнала.

**Журнализация изменений данных**

Журнализация изменений данных необходима для генерации раздела данных в пакете репликации. При помощи данного журнала осуществляется выборка тех объектов, которые претерпели изменения с момента прошедшей репликации.

Разрабатываемый компонент репликации взаимодействует с логическим уровнем метаданных, то есть раздел данных пакета репликации содержит информацию о модификации экземпляров сущностей. Журнализация изменений данных, в первую очередь, требует подписки на события добавления, изменения и удаления экземпляров сущностей.

Для структуры журнала изменений (ЖИ) можно предложить два варианта:

- Первые вариант предполагает сохранение идентификаторов атрибутов, которые были изменены при выполнении операции *UPDATE*;
- Второй вариант – это отказ от сохранения кодов атрибутов.

Их объединяет то, что нет необходимости хранить значения атрибутов измененной сущности, так как они легко восстанавливаются по искусственному ключу сущности и объекта, если обратиться к логической модели системы.

Преимуществом первого варианта является меньший размер пакета репликации, так как в него будут записаны лишь те атрибуты, которые изменились. Но он влечет за собой более сложную структуру БД для хранения журнала.

Предпочтение отдано второму способу. Структура журнала изменений в этом случае представлена на рис. 2.11.

| Первичный ключ | Идентификатор сущности | Идентификатор объекта | Код (имя) операции | Дата операции | Идентификатор транзакции | Код организации |
|---|---|---|---|---|---|---|
| 1 | IdЧеловек | IdПетров | IdAdd | 27.05.2004 | 566 | (локальная) |
| 2 | IdЧеловек | IdПетров | IdUpdate | 28.05.2004 | 567 | (локальная) |

***Рис. 2.11. Структура журнала изменений данных***

Код организации или код узла репликации, который повлек изменение объекта, необходим для того, чтобы не допустить зацикливание репликации. Предположим, что некоторые два узла ИС состоят во взаимной репликации. После процесса репликации и внесения изменений в базу-приемник, в журнал изменений данных будут внесены соответствующие записи. Нельзя допустить, чтобы при обратной репликации эти записи были обработаны и помещены в пакет, иначе они будут пересылаться между такими узлами БД бесконечно. Для предотвращения такой ситуации и вводится поле кода организации, повлекшей изменение в данных.

Размеры пакета репликации можно уменьшить, если система настроена на работу только с актуальными данными, то есть с базой данных, а не на хранилище истории. Предположим, что в БД добавили некоторый объект, и

потом его изменили (рис. 2.11). Очевидно, если репликация проводилась раньше 27.05, то в пакет репликации попадут две записи с одинаковым значением атрибутов. Различие будет лишь в проводимых операциях. Если бы в пакете отсутствовала вторая запись, то конечный результат интерпретации пакета был бы такой же, как если бы она там присутствовала. Таким образом, возможна свертка пакета. В журнале изменений можно производить аналогичные свертки.

### Журнализация изменений метаданных

Журнализация изменений метаданных требуется на этапе генерации раздела метаданных в пакете репликации. Данный журнал нужен для формирования списка сущностей и связей между ними, которые были изменены с момента последней репликации.

Журнализация требует подписки на события добавления, изменения и удаления экземпляров и связей. Такие события в ярде комплекса METAS определены не были, поэтому они были реализованы автором.

Журнал изменения метаданных представлен на рис. 2.12.

| Искусственный ключ | Имя сущности | Код (имя) операции | Дата операции | Код организации |
|---|---|---|---|---|
| 1 | Человек | IdUpdate | 13.12.2004 | (локальная) |
| 2 | Обезьяна | IdAdd | 15.12.2004 | (локальная) |

***Рис. 2.12. Структура журнала изменений метаданных***

Аналогично ЖИ данных, здесь код организации – код узла репликации, который повлек изменение сущности или связи.

### Структура пакета репликации

В качестве формата пакета репликации предлагается использовать XML-документ. Такой выбор обусловлен следующими причинами:

- простота создания XML-документа и его всесторонняя поддержка платформой программирования .NET Framework;
- возможность создания каждого XML-документа со своим набором тэгов, отвечающим специфике предметной области;
- возможность использования XML-документов в качестве универсального способа хранения данных, который включает в себя средства для разбора информации (парсеры) и средства представления ее на стороне клиента;
- поддержка XML многими браузерами, СУБД и приложениями Microsoft Office.

В общих чертах пакет репликации состоит из следующих разделов:

- *Заголовок пакета* – содержит общую информацию о пакете.
- *Метаданные* – список реплицируемых сущностей.
- *Данные* – список реплицируемых экземпляров сущностей.

Схематично структура пакета представлена на рис. 2.13.

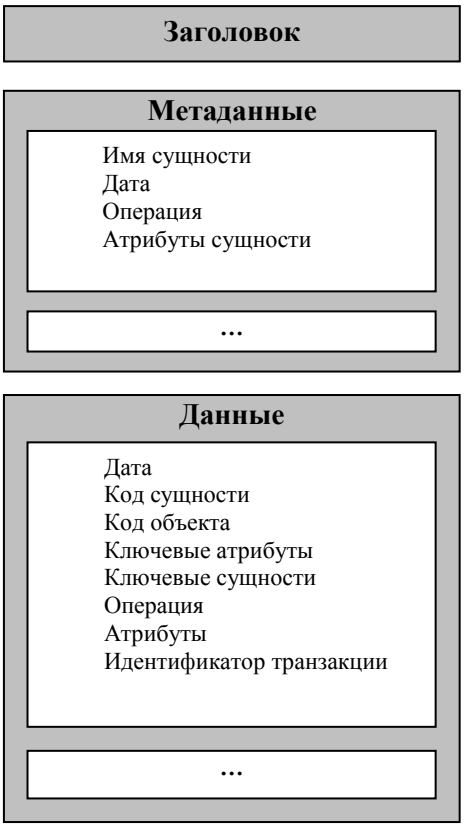


***Рис. 2.13. Структура пакета тиражирования***

Структура заголовка пакета (<*Header*>) выглядит следующим образом:

- код организации-источника;
- дата генерации пакета репликации на источнике.

В разделе метаданных (XML-тэг <*Metadata*>) содержится дата последних изменений метаданных на источнике (тэг <*LastChanged*>) и список сущностей (тэг <*Entity*>). Каждая запись сущности содержит:

- дату операции;
- код сущности;
- имя сущности;
- код операции (добавление, изменение, удаление);
- также могут быть добавлены свойства сущности:
  - описание;
  - выражение для презентационного поля;
  - атрибуты защиты: просмотр, изменение, добавление, удаление;
- список атрибутов сущности (<*Attribute*>). Для каждого атрибута:
  - названия атрибута (системное – только латинскими буквами, и презентационное, <*Name*> и <*Caption*> соответственно);
  - код типа атрибута (1 – числовой, 2 – текстовый, 3 – дата/время);
  - название типа атрибута (текстовый, целый) и т.д.;
  - обязательность наличия значения для атрибута;

- просмотр (разрешен/запрещен);
- изменение (разрешено/запрещено);
- описание атрибута;
- значение по умолчанию.

Каждая запись раздела данных (<*Data*>) содержит:

– дату операции;
– код сущности;
– код объекта;
– значение атрибутов объекта;
– информация для идентификации объекта:
  - значения ключевых атрибутов объекта;
  - код сущности, объекта и значения атрибутов ключевых сущностей;
– код операции (вставка, удаление, обновление).

### *Схема взаимодействия компонента репликации с программным обеспечением комплекса METAS*

Компонент MDK Replication взаимодействует с логическим и презентационным уровнем CASE-средства MDK Suite.

На момент начала программирования компонента, логический уровень системы содержал лишь средства для отслеживания операций над объектами сущностей. Однако для реализации реплицирования метаданных необходимо вести журнал их изменения. Также, при формировании журнала изменения данных необходимо отслеживать возникновение транзакций в системе. Поэтому возникла необходимость дополнения логического уровня комплекса.

#### Порождение события возникновения транзакции

В соответствии с моделью управления транзакциями, принятой в METAS, все компоненты системы, которые хотят организовать транзакцию, должны регистрировать SQL-команды *OleDbCommand* посредством специального класса *MDK.FoundationClasses.Context*. Поэтому очевидным стало решение порождать событие возникновения реальной СУБД-транзакции именно в этом классе. Схема возникновения события показана на рис. 2.14.

#### Порождение событий создания и удаления сущности

За построение предметной области информационной системы отвечает компонент реструктуризации. Его презентационным уровнем является компонент *DBCreator*, содержащий все формы взаимодействия с пользователем. Физическое изменение сущностей и их структур осуществляется компонентом *RestructFoundation*. В свою очередь, *RestructFoundation* взаимодействует с физическим и логическим уровнями комплекса METAS.

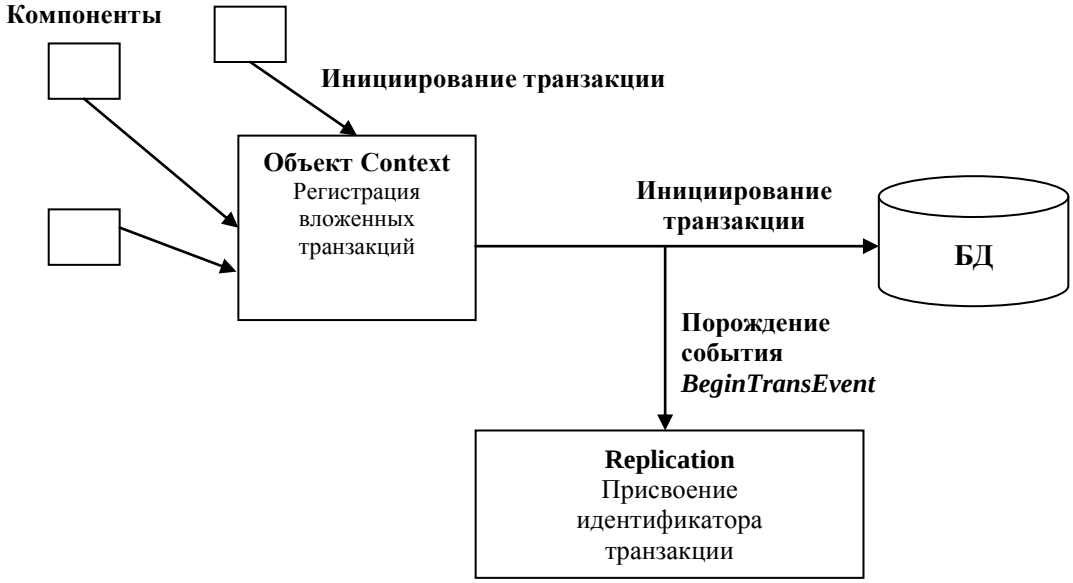


***Рис. 2.14. Схема возникновения события BeginTransEvent***

Поэтому, с точки зрения корректного объектно-ориентированного проектирования, было бы очевидным возбуждение событий создания и удаления сущностей на физическом уровне CASE-средства MDK Suite. Тем не менее, такое решение имеет некоторые сложности в смысле реализации. Дело в том, что на физическом уровне все объекты базы данных ИС (таблицы, индексы, атрибуты и т.д.) порождаются от класса *MItem*, поэтому в некоторых случаях возникают проблемы при распознавании объекта БД. Таким образом, было решено генерировать указанные события из *RestructFoundation*, но посредством логической модели (рис. 2.15).

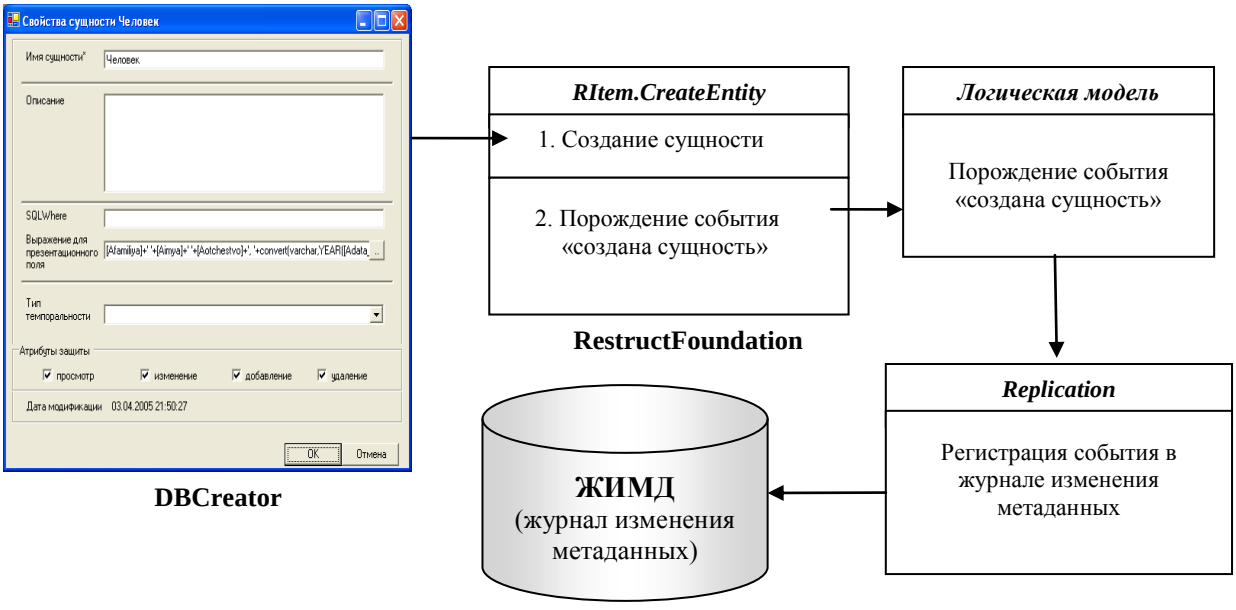


***Рис. 2.15. Схема возникновения события создания сущности***

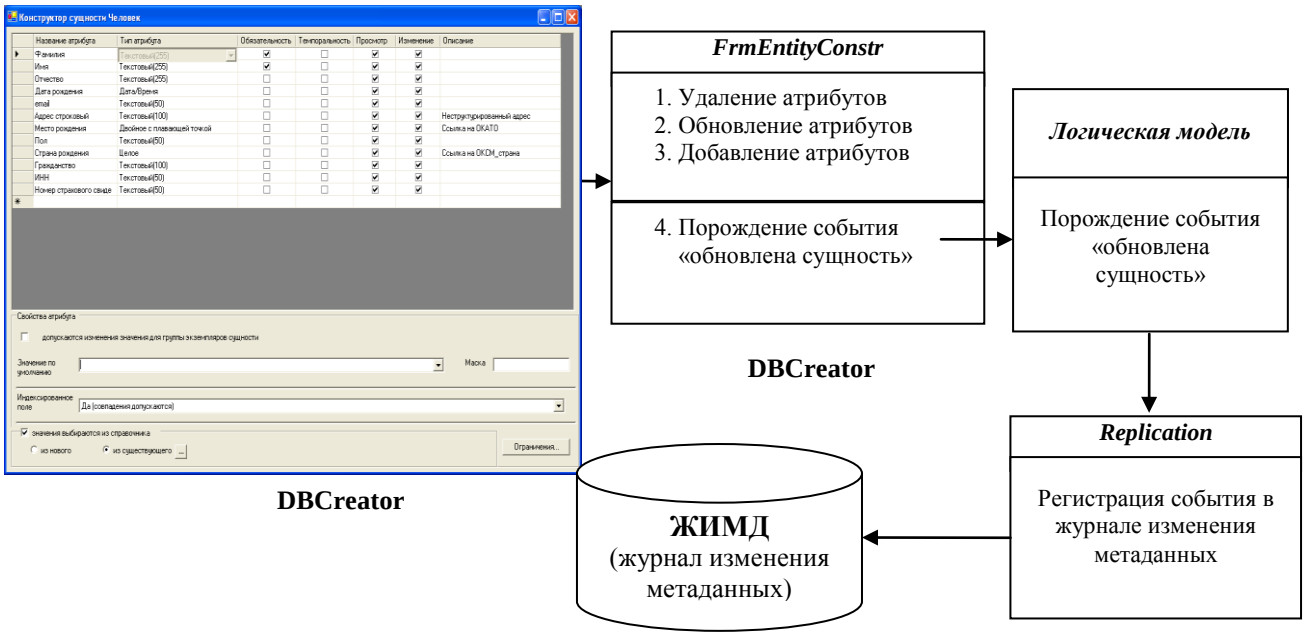


***Рис. 2.16. Схема возникновения события изменения сущности***

### Порождение события изменения сущности

Возбуждение события изменения сущности несколько отличается от событий добавления и удаления. Под изменением сущности здесь подразумевается изменение структуры сущности (списка ее атрибутов). Операции над атрибутами происходят в коде формы *FrmEntityConstr* презентационного уровня *DBCreator* (рис. 2.16).

На более низких уровнях, чем *DBCreator*, определить событие достаточно проблематично, так как на них происходят только операции над атрибутами. А изменений атрибутов при редактировании сущности может быть несколько. Нам же нужно возбуждение лишь единственного события.

### Интеграция компонента репликации с компонентом SBT Server 2.0

На данный момент репликации в системе METAS осуществляются по запросу пользователя. Вообще говоря, теоретически участие пользователя необходимо лишь при возникновении ошибочных или неоднозначных ситуаций. В будущем планируется автоматизировать процесс реплицирования (то есть организовать его без участия администратора).

Компонент SBT Server 2.0 разработан для организации автоматического (или по заданному расписанию) обмена бизнес-операциями. Поэтому он может быть использован для пересылки пакетов репликации между узлами информационной системы. Для этого необходимо помещать XML-файлы в специальную папку, а также произвести соответствующие настройки компонента.

**Интеграция компонента репликации с компонентом миграции данных**

Компонент миграции выполняет импорт данных из реляционных источников посредством механизма OLEDB в сущности логического уровня ИС. При этом можно различным образом осуществлять маппинг полей реляционной таблицы и атрибутов сущностей.

Первоначально разрабатываемый компонент репликации имел ограничение, связанное с невозможностью интерпретации данных в пакете при несовпадении метаданных на узлах репликации. Посредством интеграции с компонентом миграции оно было снято. Теперь администратор имеет возможность осуществлять импорт данных из XML-пакета, без принятия новых метаданных.

Разработанный компонент MDK Replication позволяет создавать временную таблицу «_Temp» в БД Logs. Структура указанной таблицы строится на основе подраздела <*Entity*> из раздела <*Metadata*> пакета, то есть представляет собой сущность. Названия полей соответствуют названиям атрибутов сущности, записанных на транслите (русские слова, написанные латинскими буквами). После создания структуры, система автоматически просматривает раздел данных пакета, и наполняет таблицу объектами сущностей, содержащихся в пакете.

Затем компонент репликации настраивает соответствующим образом компонент миграции и вызывает его.

## 2.4. Применяемые программные средства

Приложение является составной частью CASE-системы METAS. Программное ядро CASE-системы METAS (MDK METAS) позволяет настраиваться на различные программные платформы, работать под управлением различных операционных систем Microsoft Windows, использовать для создания информационных систем различные реляционные СУБД и источники данных, для которых существуют драйверы ODBC.

Для выполнения используется программная платформа MS .NET Framework.

CASE-технология METAS разрабатывается на платформе .NET. Компоненты тиражирования создавались с целью обеспечения средствами интеграции информационных систем, построенных с помощью указанной технологии. Поскольку они являются составной частью METAS, в качестве средства реализации также выбран Microsoft Visual Studio .NET.

Выбор платформы .NET для разработки проекта обусловлен ее очевидными преимуществами перед другими средствами и технологиями разработки

приложений. .NET – совершенно новая модель, основными преимуществами которой являются:

- Полное межъязыковое взаимодействие, в том числе межъязыковое наследование, обработка исключений и отладка.
- Общая языковая среда выполнения. Нужно отметить, что для всех языков используется один и тот же набор встроенных типов данных.
- Потенциально возможная переносимость программ на другие ОС (в случае реализации .NET Framework для них).
- Неограниченные возможности работы с существующим кодом.
- Наличие обширной библиотеки базовых классов, обеспечивающей простоту написания кода и предлагающей целостную объектную модель для всех языков программирования.

Компонент тиражирования данных об объекте будет работать более эффективно на физическом уровне метаданных ИС, то есть взаимодействуя напрямую с СУБД. Поэтому встает вопрос о выборе метода доступа к данным. Наиболее подходящей технологией является ADO.NET по следующим причинам:

- тесная интеграция с .NET;
- возможность формирования наборов для произвольного доступа к данным;
- хорошая поддержка XML;
- ADO.NET – улучшенная версия ADO для платформы .NET Framework;
- связывание таблиц в оперативной памяти.

В данной версии реализованы компоненты, использующие возможности СУБД Microsoft SQL Server, в частности, средства создания резервных копий. Это ограничивает возможности переносимости и использования продукта в гетерогенной среде.

## 2.5. Используемые технические средства и требования к аппаратуре

Требования к аппаратуре определяются требованиями, соблюдение которых необходимо для установки используемого программного обеспечения.

Минимальные требования: процессор Intel Celeron, 128 Мб ОЗУ.

## 3. Специальные условия применения и требования организационного, технического и технологического характера

Для работы с программами *нет необходимости создания специальных условий применения и выполнения особых требований организационного, технического и технологического характера*. Условия применения и требования определяются требованиями к применяемому программному и аппаратному обеспечению, перечисленными выше, а также выполнением лицензионных соглашений при использовании необходимого для работы программ программного обеспечения.

*Требования к квалификации пользователей* определяются выполняемыми ими обязанностями. Минимальные требования:

- навыки работы в среде Microsoft Windows.

*Разработчики и администраторы*, создающие схемы тиражирования и осуществляющие настройку приложения *должны удовлетворять следующим квалификационным требованиям*:

- знания и навыки в области проектирования реляционных баз данных;
- навыки администрирования реляционных СУБД.

## 4. Условия передачи программной документации или ее продажи

Продажа и передача программ и программной документации возможна на основе договора с АНО науки и образования «Институт компьютинга». Условия передачи и/или продажи полностью определяются договором, заключаемым заказчиками/покупателями с АНО «Институт компьютинга».

Представителем АНО «Институт компьютинга», имеющим право заключать договоры на передачу и/или продажу программного обеспечения, на разработки программ с использованием представленных в данном документе средств, является заместитель директора Лядова Л.Н. (e-mail: lnlyadova@mail.ru; тел.: +7 (342) 222-37-95).

## Библиографический список

1. *Артемов Д.В*. Microsoft SQL Server // Корпоративные базы данных '96: Материалы технической конференции / Центр информационных технологий. – Москва, 1996.
2. *Артемов Д.В*. Microsoft SQL Server 2000. Новейшие технологии. М.: Издательско-торговый дом «Русская редакция», 2001.
3. *Барон Г., Ладыженский Г.* Технология тиражирования данных в распределенных системах // Открытые системы. 1994, №2. С. 17–22.
4. *Вьюкова Н*. Архитектура сервера INFORMIX-OnLine Dynamic Server 7.1 и коммуникационные средства // Jet Info. 1995, Вып. 2.
5. *Карауш А.С*. Модель тиражирования библиографических баз данных // EL-Pub2003: VIII Международная конференция по электронным публикациям / Институт вычислительных технологий СО РАН. – Новосибирск, 2003.
6. *Ладыженский Г.М*. Системы управления базами данных – коротко о главном // Системы управления базами данных. 1995, №3. С. 128-136.
7. *Лядова Л.Н*. Архитектура информационной системы «Образование пермской области» // Математика программных систем: Межвуз. сб. науч. тр. / Перм. ун-т. – Пермь, 2002. – С. 25-35.
8. *Миков А.И*. Основы построения региональной распределенной информационной системы образования и науки. // Математика программных систем: Межвуз. сб. науч. тр. / Перм. ун-т. – Пермь, 2002. – С. 4-24.
9. *Мытник С.А*. Модели, методы и алгоритмы создания межсистемных информационных комплексов : дис. … канд. техн. наук : 05.13.06 – Томск, 2006 – 187 с. [http://diss.rsl.ru/diss/06/0575/060575018.pdf].
10. *Пейн К*. Освой самостоятельно ASP.NET за 21 день. М.: Издательский дом «Вильямс», 2002.
11. *Рихтер Дж*. Программирование на платформе Microsoft .NET Framework М.: Издательско-торговый дом «Русская редакция», 2002.
12. *Рыжков С.А*. Концепция метаданных в разработке информационных систем. // Математика программных систем: Межвуз. сб. науч. тр. / Перм. ун-т. – Пермь, 2002. – С.36-44.
13. *Сергеев И.В*. Программное и математическое обеспечение системы репликации данных СУБД независимых платформ: дис. … канд. техн. наук : 05.13.11 – М., 2003 – 128 с. [http://diss.rsl.ru/diss/02/0006/020006008.pdf].
14. *Сиколенко В.В*. Поддержка распределенных систем в СУБД Oracle // Системы управления базами данных. 1996, № 4. С. 27–35.
15. Служба каталогов Active Directory. // NT 5.0 или WINDOWS 2000 : справочное руководство – Гл. 2. http://www.megalib.com/books/201/2.htm

16. *Стрелков М.А*. Компонент тиражирования данных в системах, основанных на метаданных // Инженерный вестник: Научно-технический журнал общественного объединения «Белорусская инженерная академия» / Минск, 2006. – №1(21)/2.

17. *Шаврина О.В*. Разработка и реализация подхода к реплицированию данных в неоднородных распределенных средах: Дипломная работа / Пермь: Перм. ун-т., 2003.

18. Batch Queries [Electronic resource] // Sybase SQL Server Reference Manual – Chapter 5. [http://www.lcard.ru/~nail/sybase/server/58669.htm].

19. Database Language SQL Specification [Electronic resource] // Maynard, Massachusetts: Digital Equipment Corporation, 1992. [http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt].

20. *Date C.J*. What is distributed database? // InfoDB. – 1987. – №2:7.

21. Extensible Markup Language (XML) 1.0 [Electronic resource] // W3C Recommendation, 1998. [http://www.w3.org/TR/1998/REC-xml-19980210].

## СОДЕРЖАНИЕ