Глава 13

Производительность

В этой главе

- Введение
- Производительность клиент-серверного взаимодействия
- Производительность операций с базой данных
- Параметры настройки производительности
- Шаблоны кодирования для обеспечения производительности
- Средства мониторинга производительности

Введение

Зачастую команды разработчиков задумываются о производительности задним числом. Подчас ей не уделяют внимания вплоть до поздних этапов процесса разработки или, что еще хуже, до того, как уже в процессе эксплуатации продукта клиент не сообщит о серьезных проблемах с производительностью приложения. После реализации некоторой функции существенно повысить ее производительность обычно слишком сложно. Но если вы знаете, как использовать возможности оптимизации производительности в Microsoft Dynamics AX, то сможете проектировать и реализовывать модификации, обладающие оптимальной производительностью в рамках тех возможностей, которые предоставляются средой разработки и средой времени выполнения Microsoft Dynamics AX.

В этой главе обсуждаются некоторые из наиболее важных проблем оптимизации производительности, а также предлагается обзор параметров настройки производительности и средств ее мониторинга. Самую свежую информацию об оптимизации производительности Microsoft Dynamics AX вы можете найти в блоге нашей команды, которая специализируется на этих вопросах, по адресу: *http://blogs.msdn.com/axperf*. Команда регулярно обновляет свой блог, публикуя новую информацию. В этой главе мы часто ссылаемся на их отдельные публикации как на источник дополнительной информации.

Производительность клиент-серверного взаимодействия

Клиент-серверное взаимодействие является одной из ключевых областей для оптимизации производительности в Microsoft Dynamics AX. В данном разделе описываются рекомендации, общепринятые подходы и техники программирования, которые обеспечивают оптимальное клиент-серверное взаимодействие.

Уменьшения количества циклов приема-передачи

Использование следующих трех приемов в большинстве ситуаций существенно уменьшает количество циклов приема-передачи.

- Используйте метод *cacheAddMethod* для всех подходящих *display*- и *edit*-методов на формах вместе с декларативным кэшированием *display*-методов.
- Выполните рефакторинг классов *RunBase* с целью поддержки маршализации диалога между сервером и клиентом.
- Используйте подходящие техники создания табличных индексов и кэширования табличных данных.

Метод *cacheAddMethod*

Display- и *edit*-поля используются на экранных формах для отображения данных, которые вычисляются на основании других полей таблицы. Методы для вычисления значения данных полей можно создавать на таблицах и на формах. По умолчанию значения данных полей рассчитываются одно за другим; при необходимости обращения к серверу в одном из методов, что обычно и случается, каждый из этих методов обращается к серверу самостоятельно. Значения данных полей пересчитываются каждый раз при обновлении данных экранной формы, а это может происходить, когда пользователь изменяет значений полей, использует пункты меню или просто нажимает F5. Соответственно, использование такого подхода является дорогостоящей операцией, как из-за большого количества циклов приема-передачи между клиентом и сервером приложения (AOS), так и из-за количества запросов к базе данных, посылаемых при этом сервером приложения.

Кэширование *display*- и *edit*-методов, объявленных на источнике данных формы, невозможно, так как методам требуется доступ к метаданным

экранной формы. При возможности такие методы следует перенести на уровень таблицы. Для кэширования значений *display*- и *edit*-методов, объявленных на таблице, используется метод *FormDataSource.cacheAddMethod*. Он позволяет движку форм рассчитать значения всех требуемых полей за один цикл приема-передачи на сервер, а затем закэшировать результаты этого расчета. Для использования метода *cacheAddMethod* необходимо в методе *init* источника данных формы, который использует *display*- или *edit*-методы, вызвать его на объекте источника данных, передав в качестве параметра название *display*- или *edit*-метода. Пример использования этого метода можно найти на источнике данных *SalesLine* формы *SalesTable*. В методе *init* этого источника данных присутствует следующий код:

```
public void init()
{
    super();
    salesLine_ds.cacheAddMethod(tableMethodStr(SalesLine, invoicedInTotal), false);
    salesLine_ds.cacheAddMethod(tableMethodStr(SalesLine, deliveredInTotal), false);
    salesLine_ds.cacheAddMethod(tableMethodStr(SalesLine, itemName), false);
    salesLine_ds.cacheAddMethod(tableMethodStr(SalesLine, timeZoneSite), true);
}
```

Если вышеприведенный код закомментировать, то каждый из *display*-методов будет выполняться для всех операций с источником данных, что увеличит количество обращений к AOS, а также количество запросов к базе данных. Более подробную информацию о *cacheAddMethod* вы можете найти по адресу: *http://msdn.microsoft.com/en-us/library/formdatasource.cacheaddmethod.aspx*.

Примечание. Не кэшируйте приведенным способом *display*- или *edit*-методы, которые не используются на форме. Такие методы будут вычисляться для каждой записи, в то время как их результаты отображены не будут.

В Microsoft Dynamics AX 2009 компания Microsoft вложила много усилий в инфраструктуру *cacheAddMethod*. В предыдущих версиях системы данный подход работал только для *display*-методов и только при загрузке формы. Начиная с Microsoft Dynamics AX 2009, кэширование используется как для *display*-, так и для *edit*-методов, причем на протяжении всего жизненного цикла экранной формы, в том числе при вызове методов *reread*, *write* и *refresh*. При вызове любого из этих методов происходит обновление

значений *display*- и *edit*-полей, однако теперь ядро обновляет все поля сразу, а не поочередно. В Microsoft Dynamics AX 2012 к этим возможностям добавлена еще одна – декларативное кэширование *display*-методов.

Декларативное кэширование *display*-методов

Чтобы воспользоваться возможностью декларативного кэширования *display*-методов, установите свойство *CacheDataMethod* элемента управления формы в значение *Yes*. Свойство *CacheDataMethod* показано на рис. 13-1.

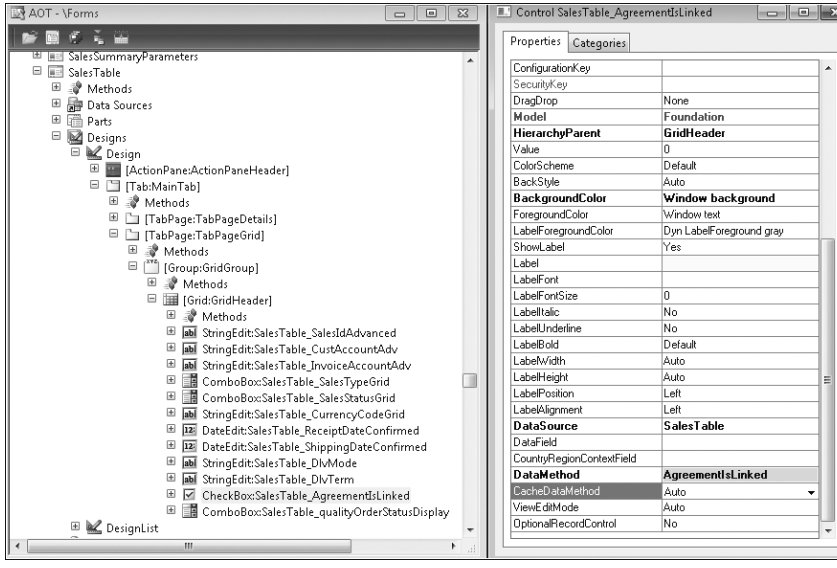

Рис. 13–1. Свойство *CacheDataMethod*

Для нового свойства доступны значения *Auto*, *Yes* и *No*. По умолчанию используется значение *Auto*, которое приравнивается к *Yes*, если метод реализован на источнике данных формы, доступном только для чтения. В основном это относится к формам списков. Если один и тот же метод привязан к нескольким элементам управления на форме, то он будет закэширован, если хотя бы для одного элемента управления значение свойства можно будет приравнять к *Yes*.

Прием с классами *RunBase*

Классы *RunBase* составляют каркас для выполнения большей части бизнес-логики в Microsoft Dynamics AX. *RunBase* обеспечивает большую часть базовой функциональности, необходимой для выполнения деловой операции: отображение диалогового окна для ввода данных, выполнение бизнес-логики приложения, а также выполнение операций в пакетном режиме.

Примечание. В Microsoft Dynamics AX 2012 появилась инфраструктура *SysOperation*, которая предоставляет большую часть функциональности инфраструктуры *RunBase* и в конечном итоге заменит ее. Более детальную общую информацию об инфраструктуре *SysOperation* вы можете найти в главе 14 . Более детальную информацию об оптимизации производительности при использовании инфраструктуры *SysOperation* вы можете найти в разделе «Инфраструктура SysOperation» далее в этой главе.

При выполнении бизнес-логики приложения через инфраструктуру *RunBase* процесс обработки выглядит, как показано на рис. 13-2.

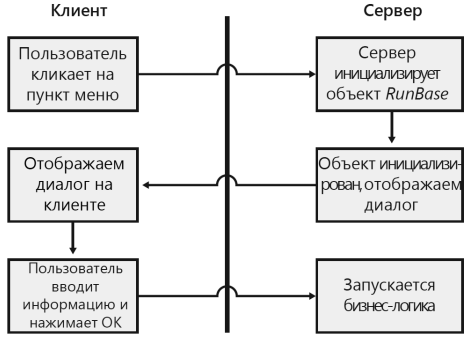

Рис. 13–2. Клиент-серверное взаимодействие в инфраструктуре *RunBase*

Большинство проблем с количеством циклов приема-передачи в инфраструктуре *RunBase* связано с использованием диалогового окна. Код класса *RunBase* следует выполнять на сервере приложения из соображений безопасности, а также потому, что код обработки обычно выполняет большое количество запросов к базе данных. Однако при указании на самом классе *RunBase*, что он должен выполняться на сервере приложения, возникает проблема. При выполнении класса *RunBase* на сервере диалого-

вое окно для ввода данных также создается и управляется с сервера приложения, что приводит к огромному количеству клиент-серверных вызовов.

Во избежание излишних клиент-серверных вызовов, отметьте класс *RunBase* для выполнения на том уровне, откуда он вызван (*Called From*), после чего укажите, что метод *construct* класса либо пункт меню, с помощью которого вызывается данный класс, должны выполняться на сервере приложения.

Выполнение с использованием *Called From* позволяет инфраструктуре *RunBase* упаковывать и передавать объект класса между клиентом и сервером, избавляясь от необходимости управления диалоговым окном для ввода данных с сервера приложения, что в значительной мере уменьшает количество клиент-серверных вызовов. Следует помнить, что для успешного выполнения сериализации объекта класса необходимо соответствующим образом реализовать методы *pack* и *unpack*.

За более подробными инструкциями по оптимальной реализации классов *RunBase* с точки зрения количества клиент-серверных вызовов обратитесь к техническому документу «Microsoft Dynamics AX 2009 White Paper: *RunBase* Patterns», доступному по адресу: *http://www.microsoft.com/en-us/download/details.aspx?id=19517*.

Кэширование и индексация

В Microsoft Dynamics AX реализована инфраструктура кэширования данных на уровне приложения клиента, которая позволяет существенно сократить количество обращений за данными к серверу приложения. Это кэширование данных работает по любому уникальному индексу таблицы, соответственно, при доступе к данным из клиентского кода следует по возможности использовать существующий уникальный ключ записи. Также следует проверить, что в репозитарии прикладных объектов (AOT) все уникальные индексы отмечены как таковые. С помощью инструмента проверки на соответствие рекомендациям можно убедиться, что на всех таблицах приложения существует первичный ключ. Более подробную информацию об инструменте проверки на соответствие рекомендациям вы можете найти в главе 2.

Для того чтобы использовать кэширование на клиенте, необходимо соответственным образом установить значение свойства *CacheLookup* на каждой таблице. В табл. 13-1 приводится перечень допустимых значений этого свойства. Каждое значение подробно обсуждается в разделе «Кэширование» далее в этой главе.

Табл. 13–1. Значения свойства *CacheLookup*

Настройка кэширования	Описание
Found	При выполнении запроса к таблице по первичному или другому уникальному ключу выбранная запись помещается в кэш до завершения текущего сеанса Microsoft Dynamics AX или до обновления этой записи. В случае обновления данной записи другим сервером приложения будет очищен кэш на всех остальных серверах. Это значение свойства следует указывать для основных данных системы
NotInTTS	Результат аналогичен использованию значения *Found* за исключением случаев, когда выборка данных происходит в области действия транзакции базы данных. В таких случаях кэш записи очищается, и запрос отправляется на уровень базы данных. Это значение свойства следует указывать для транзакционных таблиц системы
FoundAndEmpty	Результат аналогичен использованию *Found* за исключением ситуации, когда в результате запроса не возвращается ни одной записи. В таком случае в кэш сохраняется информация об отсутствии такой записи таблицы. Это значение свойства следует указывать для основных данных, привязанных к определенной стране/региону, а также для основных данных, которые в рабочем режиме могут отсутствовать в системе
EntireTable	На сервере приложения все данные таблицы помещаются в кэш, а на клиенте это значение обрабатывается как *Found*. Данное значение свойства следует указывать для таблиц, в которых количество записей ограничено и заранее известно, как в случае с таблицами параметров модуля
None	Кэширование данных не выполняется. Это свойство следует использовать только в некоторых особых случаях, таких как необходимость отключения оптимистичной блокировки на таблицах

Кэширование по индексу работает только в том случае, когда в выражении *where* запроса указаны поля уникального ключа таблицы. Новой возможностью по сравнению с предыдущими версиями является использование кэширования данных для запросов с объединением таблиц по полям уникального индекса; эта возможность обсуждается далее в этой главе в разделах «Кэш для запросов с объединением таблиц по полям уникального индекса» и «Производительность операций с базой данных».

Этот кэш поддерживает только отношения 1:1, иными словами, данные из кэша не будут использоваться, если в запросе есть объединение таблиц с отношениями 1:*n* или если запрос выполняется по данным нескольких компаний. В Microsoft Dynamics AX 2012 поддерживается выборка данных из кэша для любых комбинаций условий фильтрации выборки, если среди них встречается набор условий фильтрации по полям уникального индекса.

При использовании кэширования *EntireTable* все содержимое таблицы хранится на сервере приложения, но для приложения клиента в данном случае используется кэширование *Found*. Для таблиц, которые содержат только одну строку в каждой компании, таких как таблицы параметров, добавляйте ключевое поле с заранее известным значением, таким как 0. Это позволит приложению клиента использовать кэш при доступе к записям таких таблиц. Примером использования ключевого поля в стандартном приложении Microsoft Dynamics AX является таблица *CustParameters*.

Написание кода, который учитывает нюансы клиент-серверного взаимодействия

При написании кода приложения следует учитывать то, где будет исполняться данный код и где находятся объекты, которые он использует. Вот некоторые аспекты, которые следует иметь в виду.

- Объекты, чье свойство *RunOn* установлено в *Server*, всегда создаются на сервере.
- Объекты, чье свойство *RunOn* установлено в *Client*, всегда создаются на клиенте.
- Объекты, чье свойство *RunOn* установлено в *Called from*, создаются на том уровне, где используется класс.

Одно замечание: классы, для которых отмечено исполняться на уровне клиента либо на уровне сервера, не удастся сериализовать на другой слой с помощью методов *pack/unpack*. При попытке сериализации такого серверного объекта на клиенте вы просто получите новый объект на сервере с такими же значениями, как и исходный объект. Статические методы выполняются на том слое, который указан в их определении с помощью ключевых слов *Client*, *Server* или *Client Server*.

Корректно используйте временные таблицы типа *inMemory*

Временные таблицы являются частым источником как обращений сервера к клиенту, так и излишних обращений клиента к серверу. В отличие от обычных буферов таблиц, временные таблицы находятся на том слое приложения, на котором в них была вставлена первая запись. К примеру, если временная таблица объявлена на сервере, а вставка первой записи в эту таблицу произошла на клиенте, в то время как вставка всех последующих записей проходила на уровне сервера, любое обращение к этой таблице на сервере будет обрабатываться через слой клиента.

Заполнять временные таблицы обычно лучше на сервере, так как данные, вставляемые во временную таблицу, скорее всего, выбираются из базы данных. Тем не менее следует быть осторожным, если вам нужно обработать эти данные и вывести их на экранной форме. Для оптимального выполнения этой задачи следует заполнить временную таблицу данными на сервере приложения, выполнить ее сериализацию в контейнер, который затем передать на уровень клиента, где считать все данные из контейнера в клиентскую временную таблицу.

По возможности всегда избегайте объединения временных таблиц типа *inMemory* с обычными таблицами базы данных, потому что в этом случае AOS сначала выберет все данные для текущей компании из таблицы базы данных, а уже затем начнет объединять их в памяти с данными временной таблицы, что является очень ресурсоемкой и медленной операцией. По возможности избегайте кода, подобного приведенному в следующем примере:

```
public static server void ImMemTempTableDemo()
{
    RealTable rt;
    InMemTempTable tt;
    int i;

    // Заполнение временной таблицы
    ttsBegin;
    for (i=0; i<1000; i++)
    {
        tt.Value = int2str(i);
        tt.insert();
    }
    ttsCommit;
    // Неэффективное объединение с обычной таблицей базы данных. Если временная
```

```
    // таблица имеет тип inMemory, то такое объединение приведет к отправке 1000
    // запросов select к таблице базы данных и, таким образом, будет выполнена 1000
    // циклов приема-передачи между сервером приложения и СУБД.

    select count(RecId) from tt join rt where tt.value == rt.Value;
    info(int642str(tt.Recid));
}
```

Если вы решите использовать временные таблицы типа *inMemory*, то существенно улучшить производительность запросов к ним позволит создание индексов, подходящих для тех запросов, которые вы планируете по ним выполнять. По сравнению с созданием индексов для запросов по обычным таблицам тут есть одно отличие: поля индекса должны быть в том же порядке, в каком они идут в запросе. Например, следующий запрос существенно выиграет от наличия на таблице *TmpDimTransExtract* индекса по полям *AccountMain*, *ColumnId* и *PeriodCode*:

```
SELECT SUM(AmountMSTDebCred) FROM TmpDimTransExtract WHERE
((AccountMain>=N'11011201' AND AccountMain<=N'11011299')) AND ((ColumnId = 1))
AND ((PeriodCode = 1))
```

Используйте временные таблицы типа *TempDB*

Временные таблицы типа *inMemory* вы можете легко заменить таблицами типа *TempDB*, у которых есть следующие преимущества над таблицами типа *inMemory*.

- Вы можете эффективно использовать соединения таблиц базы данных с таблицами типа *TempDB*.
- Вы можете использовать операторы, работающие с наборами данных, для заполнения таблиц типа *TempDB*, сокращая число циклов приема-передачи между сервером приложения и СУБД.

Чтобы создать временную таблицу типа *TempDB*, установите свойство *TableType* таблицы в значение *TempDB*, как показано на рис. 13-3.

Совет. Даже если временные таблицы не удаляются полностью, а усекаются и используются повторно, как только выйдут из области видимости, старайтесь минимизировать число временных таблиц, которые нужно создать. Создание временной таблицы сопряжено с определенными накладными расходами, так что используйте их только в случае реальной необходимости.

Рис. 13–3. Используйте свойство *TableType* для создания временной таблицы типа *TempDB*

Если вы используете временные таблицы типа *TempDB*, не заполняйте их построчно, как в примере ниже.

```
public static server void SQLTempTableDemo1()
{
    SQLTempTable tt;
    int i;

    // Заполнение временной таблицы; это приведет к 1,000 циклов приема-передачи
    // между сервером приложений и базой данных

    ttsBegin;
    for (i=0; i<1000; i++)
    {
        tt.Value = int2str(i);
        tt.insert();
    }
    ttsCommit;
}
```

Вместо этого используйте операторы, работающие с наборами данных. В следующем примере показано, как использовать такие операторы для создания эффективных объединений с постоянной таблицей базы данных.

```
public static server void SQLTempTableDemo2()
{
    RealTable rt;
    SQLTempTable tt;

    // Заполнение временной таблицы с помощью всего лишь одного цикла
    // приема-передачи между сервером приложений и базой данных.

    ttsBegin;
    insert_recordset tt (Value)
        select Value from rt;
    ttsCommit;
```

```
    // Эффективное соединение с постоянной таблицей базы данных приводит лишь к
    // одному циклу приема-передачи между сервером приложений  и базой данных.
    // Если бы временная таблица была типа inMemory, это объединение привело бы к
    // выполнению 1,000 операторов select к постоянной таблице.

    select count(RecId) from tt join rt where tt.value == rt.Value;
    info(int642str(tt.Recid));
}
```

Устранение обратных клиентских вызовов

Клиентские обратные вызовы (Callback) происходят в случаях, когда клиент вызывает метод, который выполняется на сервере, а сервер, в свою очередь, при выполнении этого метода вызывает клиентский метод. Это случается по двум причинам. Во-первых, такое происходит, когда приложение клиента посылает недостаточное количество информации на сервер приложения при вызове метода или посылает на сервер клиентский объект, инкапсулирующий эту информацию. Во-вторых, такое случается, когда с уровня сервера приложения происходит обновление или другое обращение к экранной форме.

Для устранения вызовов первого типа убедитесь, что при вызове серверного метода в него передается вся необходимая информация в сериализованном виде. К примеру, можно передавать данные, упакованные в контейнер, буферы записей, а также переменные значимых типов (то есть типы *int*, *str*, *real*, *boolean* и др.). При обращении к таким типам серверу не требуется обращаться на уровень клиента, как в случае с объектными типами.

Для устранения вызовов второго типа передавайте в серверный метод всю необходимую информацию об экранной форме и выполняйте обновление формы только по завершении выполнения серверного метода, вместо того чтобы делать это с уровня сервера приложения. Оптимальным способом для того, чтобы отложить выполнение действий на уровне клиента, является использование сериализации с помощью методов *pack*/*unpack*. С использованием такого подхода объект класса можно упаковать в контейнер и распаковать данные уже на другой стороне вызова.

Объединение вызовов

Для обеспечения минимального количества клиент-серверных вызовов их следует объединять в один вызов статического серверного метода, в который передать всю необходимую для выполнения обработки информацию. В качестве примера можно рассмотреть статический серверный метод *NumberSeq*::*getNextNumForRefParmId*, который содержит следующий код:

```
return NumberSeq::newGetNum(CompanyInfo::numRefParmId()).num();
```

Если бы этот код вызывался с уровня клиента, то это привело бы к возникновению четырех циклов удаленного вызова процедур (remote procedure call, RPC): один цикл для метода *newGetNum*, один – для *numRefParmId*, один – для метода *num* и один – для удаления из памяти объекта *NumberSeq*, созданного в результате. Благодаря использованию статического серверного метода, эта операция выполняется за один клиент-серверный вызов.

Еще одним распространенным примером, в котором следует объединять вызовы в один, является выполнение операций работы с транзакциями базы данных на уровне клиента. Часто разработчики пишут код, подобный приведенному ниже.

```
ttsBegin;
record.update();
ttsCommit;
```

Можно избавиться от двух лишних циклов приема-передачи, если объединить вышеприведенный код в один серверный метод. Все операции с транзакциями базы данных выполняются только на уровне сервера. Соответственно, не следует вызывать *ttsbegin* и *ttscommit* с уровня клиента для начала транзакции на сервере базы данных, если уровень *ttslevel* равен 0.

Передача табличных буферов по значению вместо передачи по ссылке

Методы класса *global buf2con* и *con2buf* используются в коде X++ для преобразования табличных буферов в контейнеры и обратно. В эти методы была добавлена новая функциональность, и они стали выполняться намного быстрее, чем в предыдущих версиях Microsoft Dynamics AX.

Преобразование табличных буферов в контейнеры может быть полезным, если вам нужно передать табличный буфер между различными уровнями (например, между клиентом и сервером). Передача контейнера предпочтительнее передачи табличного буфера, потому что контейнеры передаются по значению, а табличные буферы – по ссылке. Передача объектов по ссылке между уровнями приводит к большому числу RPC-вызовов и ухудшает производительность приложения. Каждый вызов экземплярного метода объекта, созданного на другом уровне, приводит к RPC-вызову. Для улучшения производительности вы можете избавиться от таких обратных вызовов за счет создания локальных копий табличных буферов, используя *buf2con,* чтобы упаковать их, и *con2buf* – чтобы распаковать.

В следующем примере приводится форма, выполняющаяся на клиенте и передающая данные для обновления на сервер. В примере показано, как эффективно передать табличный буфер с использованием минимального количества RPC-вызовов.

Примечание. На практике вы обычно будете работать не с данными временной таблицы, а с реальными данными БД.

```
public void updateResultField(Buf2conExample _clientRecord)
{
    container packedRecord;

    // упаковка записи перед отправкой на сервер

    packedRecord = buf2Con(_clientRecord);

    // отправка упакованной записи на сервер и получение контейнера с результатом

    packedRecord = Buf2ConExampleServerClass::modifyResultFromPackedRecord(packedR
ecord);

    // распаковка полученного в результате контейнера в табличный буфер на клиенте

    con2Buf(packedRecord, _clientRecord);
    Buf2conExample_ds.refresh();
}
```

Изменение данных на сервере и затем отправка контейнера обратно:

```
public static server container modifyResultFromPackedRecord(container _packedRecord)
{
    Buf2conExample recordServerCopy = con2Buf(_packedRecord);
    Buf2ConExampleServerClass::modifyResult(recordServerCopy);
    return buf2Con(recordServerCopy);
}
public static server void modifyResult(Buf2conExample _clientTmpRecord)
{
    int n = _clientTmpRecord.A;
    _clientTmpRecord.Result = 0;
    while (n > 0)
    {
        _clientTmpRecord.Result = Buf2ConExampleServerClass::add(_clientTmpRecord);
        n--;
    }
}
```

Производительность операций с базой данных

В предыдущем разделе основное внимание уделялось ограничению объема данных, передаваемых между уровнями клиента и сервера, но эти уровни – всего лишь два из трех, которые задействованы при исполнении приложения Microsoft Dynamics AX. Последний уровень – уровень базы данных, оптимизация обмена пакетами данных между уровнем сервера и уровнем базы данных – также очень важен. В данном разделе внимание уделяется оптимизации исполнения прикладной логики, которая взаимодействует с базой данных. Среда времени выполнения Microsoft Dynamics AX помогает минимизировать число вызовов от уровня сервера к уровню базы данных за счет поддержки DML-операторов на базе наборов и кэширования данных. Однако при написании кода разработчик также должен уделять внимание объему данных, отправляемых серверу приложения с уровня базы данных. Чем меньше данных передается, тем быстрее завершится выборка данных из базы и, соответственно, уменьшится количество передаваемых по сети пакетов, что в итоге приведет к уменьшению объема используемой памяти. Все эти усилия приведут к ускорению исполнения прикладной логики, а это, в свою очередь, приведет к сокращению длительности транзакций базы данных, сокращению количества и продолжительности блокировок и улучшению параллелизма и пропускной способности.

Примечание. Вы можете еще больше повысить производительность операций с базой данных за счет определенной реализации прикладной логики. Например, обеспечивая один и тот же порядок изменения различных таблиц и записей, вы избежите клинчей и снизите количество исключений, обработка которых требует повторного выполнения определенных участков кода. Тратя время до начала транзакций на их подготовку, чтобы сделать их настолько короткими, насколько возможно, вы сократите время, в течение которого удерживаются блокировки, и за счет этого в конечном итоге улучшите параллелизм выполнения транзакций. Также большое значение имеют факторы, связанные с проектированием схемы данных, такие как создание и использование необходимых индексов. Однако эти темы выходят за рамки данной книги.

Операторы манипуляции данных на основе наборов

Язык X++ содержит специфические операторы и классы, выполняющие манипуляции с данными базы данных на основе набора записей. Конструкции на основе наборов обладают преимуществом по сравнению с конструкциями на основе записей – они требуют значительно меньше обращений сервера приложений к базе данных. В следующем примере кода X++, в котором выполняется выбор нескольких записей таблицы *CustTable* и обновление в каждой из них значения поля *CreditMax*, иллюстрируется необходимость одного цикла приема-передачи к базе данных для выбора записей и по одному циклу для исполнения каждого оператора *update*.

```
static void UpdateCustomers(Args _args)
{
    CustTable custTable;

    ttsBegin;

    while select forupdate custTable
        where custTable.CustGroup == '20' // обращение к базе данных
    {
        custTable.CreditMax = 1000;
        custTable.update(); // обращение к базе данных
    }

    ttsCommit;
}
```

Если под условия обновления подходят 100 записей таблицы *CustTable*, поскольку значение поля *CustGroup* для них равно '20', число обращений к базе данных составит 101 (1 обращение для выполнения оператора *select* и 100 обращений для выполнения каждого оператора *update*). Число обращений для выполнения оператора *select* в действительности может быть несколько больше, в зависимости от числа записей таблицы *CustTable*, которые могут быть одновременно извлечены из базы данных и переданы серверу приложения.

Теоретически вы могли бы переписать код из предыдущего примера так, чтобы выполнить обновление всех записей за один вызов к серверу базы данных, как показано в примере ниже. Здесь демонстрируется ис-

пользование оператора *update_recordset* для отправки базе данных всего лишь одного SQL-оператора *UPDATE*.

```
static void UpdateCustomers(Args _args)
{
    CustTable custTable;

    ttsBegin;

    update_recordset custTable setting CreditMax = 1000
        where custTable.CustGroup == '20'; // одно обращение к базе данных
    ttsCommit;
}
```

Однако в силу нескольких причин использование буфера записей именно таблицы *CustTable* не приведет к одному-единственному обращению к базе данных. Эти причины описываются в последующих разделах для каждой из конструкций на основе наборов, поддерживаемых средой времени выполнения Microsoft Dynamics AX. В этих разделах также описываются методы, которые позволят вам модифицировать предыдущий сценарий так, чтобы гарантировать, что выполняется лишь одно обращение к базе данных, даже если используется буфер записей таблицы *CustTable*.

Важно. Описанные далее операторы на основе наборов не улучшают производительность при работе с временными таблицами типа *inMemory*. Среда времени выполнения Microsoft Dynamics AX для временных таблиц типа *inMemory* всегда понижает уровень оператора на основе наборов к оператору на основе записей. Это справедливо вне зависимости от того, как таблица стала временной (указано ли это в метаданных в свойствах таблицы, отключена ли таблица с помощью конфигурационных ключей приложения Microsoft Dynamics AX или же она явно определена как временная в коде X++). Также обратите внимание, что понижение средой разработки уровня операции для временных таблиц типа *inMemory* приводит к вызову методов *doInsert*, *doUpdate* и *doDelete* буфера записей, а соответственно прикладная логика в перекрытых методах не вызывается.

Операторы манипуляции данных на основе наборов и наследование таблиц

Оператор манипуляции данных на основе наборов, такой как *insert_recordset*, *update_recordset* или *delete_from*, не понижается до операции на основе записей в случае родительской или дочерней таблицы в иерархии наследования, если только не выполнено условие, из-за которого такое понижение необходимо выполнить. Операторы *insert_recordset* и *update_recordset* могут обновлять или вставлять все соответствующие записи в указанную таблицу и все дочерние или родительские таблицы, но не в какие-либо из производных таблиц. Оператор *delete_from* обрабатывается иначе, потому что он удаляет все соответствующие записи из текущей таблицы и ее дочерних и родительских таблиц, чтобы гарантировать, что запись полностью удалена из базы данных. Более подробную информацию об условиях, при которых операторы на основе наборов понижаются до уровня отдельных записей, вы найдете в следующих разделах.

Оператор *insert_recordset*

Оператор *insert_recordset* позволяет выполнить вставку нескольких записей в таблицу за одно обращение к базе данных. В следующем примера кода иллюстрируется использование этого оператора для копирования записей *InventSum* для одной номенклатуры во временную таблицу для последующего использования.

```
static void CopyItemInfo(Args _args)
{
    InventTable                 inventTable;
    InventSum                   inventSum;
    InsertInventTableInventSum  insertInventTableInventSum;

    // insert_recordset использует лишь одно обращение к БД для операции копирования
    // оператор insert, использующий отдельные записи, потребовал бы отдельного
    // обращения к БД на каждую запись из InventSum

    ttsBegin;
    insert_recordset insertInventTableInventSum (ItemId,AltItemId,PhysicalValue,PostedValue)
        select ItemId,AltItemid from inventTable where inventTable.ItemId == '1001'
        join PhysicalValue,PostedValue from inventSum
        where inventSum.ItemId == inventTable.ItemId;
    ttsCommit;
    select count(RecId) from insertInventTableInventSum;
    info(int642str(insertInventTableInventSum.RecId));
```

```
    // далее следует код, использующий скопированные данные...
}
```

Обращение к базе данных включает в себя выполнение трех операторов.

1. Часть *select* оператора *insert_recordset* выполняется, когда выбранные записи вставляются в новую временную таблицу в БД. Синтаксис оператора *select* при выполнении в Transact-SQL схож с *SELECT <список полей> INTO <временная таблица> FROM <исходные таблицы> WHERE <предикаты>*.
2. Записи из временной таблицы вставляются напрямую в целевую таблицу с использованием синтаксиса наподобие *INSERT INTO <целевая таблица> (<список полей>) SELECT <список полей> FROM <временная таблица>*.
3. Временная таблица удаляется с помощью выполнения оператора *DROP TABLE <временная таблица>*.

Такой подход дает огромное преимущество в плане производительности по сравнению со вставкой по одной записи, показанной в следующем примере кода X++, который решает ту же задачу.

```
static void CopyItemInfoLineBased(Args _args)
{
    InventTable                 inventTable;
    InventSum                   inventSum;
    InsertInventTableInventSum  insertInventTableInventSum;

    ttsBegin;
    while select ItemId,AltItemid from inventTable where inventTable.ItemId == '1001'
        join PhysicalValue,PostedValue from inventSum
        where inventSum.ItemId == inventTable.ItemId
    {
        InsertInventTableInventSum.ItemId = inventTable.ItemId;
        InsertInventTableInventSum.AltItemId = inventTable.AltItemId;
        InsertInventTableInventSum.PhysicalValue = inventSum.PhysicalValue;
        InsertInventTableInventSum.PostedValue = inventSum.PostedValue;
        InsertInventTableInventSum.insert();
    }
ttsCommit;

select count(RecId) from insertInventTableInventSum;
```

```
    info(int642str(insertInventTableInventSum.RecId));

    // далее следует код, использующий скопированные данные...
}
```

Если в InventSum содержатся 10 записей, для которых *ItemId* равен *1001*, то в этом сценарии потребуется одно обращение к БД для выполнения оператора *select* и дополнительно 10 обращений к БД для вставки записей, что в итоге дает 11 обращений к БД.

Оператор *insert_recordset* может быть понижен из оператора на основе наборов до оператора на основе отдельных записей, если выполняется любое из следующих условий.

- Таблица кэшируется с использованием настройки *EntireTable*.
- На целевой таблице перекрыт метод *insert* или метод *aosValidateInsert*.
- На вставке записей в целевую таблицу настроено срабатывание оповещений.
- На вставку записей в целевую таблицу настроено логирование в журнал БД.
- На целевой таблице настроена безопасность на уровне записей (RLS). Если RLS настроена только на исходной таблице или таблицах, то *insert_recordset* не понижается до операций на основе отдельных записей.
- Свойство таблицы *ValidTimeStateFieldType* установлено в значение, отличное от *None*.

Среда времени выполнения Microsoft Dynamics AX автоматически обрабатывает понижение и внутренне выполняет логику, схожую с циклом *while select*, показанным в предыдущем примере.

Важно. Когда среда времени выполнения Microsoft Dynamics AX проводит проверку на наличие перекрытых методов, она лишь проверяет наличие таких методов, но не проверяет, содержат ли перекрытые методы лишь код X++ по умолчанию. Поэтому метод рассматривается как перекрытый, даже если он содержит лишь следующий код X++:

```
public void insert()
{
    super();
}
```

При наличии такого метода любой оператор *insert*, работающий на основе наборов данных, понижается до уровня работы с отдельными записями.

Если уровень кэширования таблицы не установлен в *EntireTable*, то вы можете избежать понижения уровня оператора, вызываемого прочими упомянутыми ранее причинами. Табличный буфер содержит методы, отключающие проверки, осуществляемые средой времени выполнения для определения того, надо ли понижать уровень работы *insert_recordset* до отдельных записей.

- Вызов *skipDataMethods(true)* отключает проверку, определяющую, перекрыт ли метод *insert*.
- Вызов *skipAosValidation(true)* отключает проверку в методе *aosValidateInsert*.
- Вызов *skipDatabaseLog(true)* отключает проверку, определяющую, настроено ли логирование вставок в таблицу в журнал БД.
- Вызов *skipEvents(true)* отключает проверку, определяющую, настроены ли на вставку в таблицу какие-либо уведомления.

Следующий код X++, включающий вызов *skipDataMethods(true)*, обеспечивает то, чтобы оператор *insert_recordset* не был понижен до уровня работы с отдельными записями из-за перекрытого метод *insert* на таблице *InsertInventTableInventSum*.

```
static void CopyItemInfoskipDataMethod(Args _args)
{
    InventTable                 inventTable;
    InventSum                   inventSum;
    InsertInventTableInventSum  insertInventTableInventSum;

    ttsBegin;

    // пропуск проверки на перекрытый метод insert.

    insertInventTableInventSum.skipDataMethods(true);
    insert_recordset insertInventTableInventSum (ItemId,AltItemId,PhysicalValue,PostedValue)
        select ItemId,Altitemid from inventTable where inventTable.ItemId == '1001'
        join PhysicalValue,PostedValue from inventSum
        where inventSum.ItemId == inventTable.ItemId;
    ttsCommit;
```

```
    select count(RecId) from insertInventTableInventSum;
    info(int642str(insertInventTableInventSum.RecId));

    // далее следует код, использующий скопированные данные...
}
```

Важно. Используйте данные *skip*-методы крайне осторожно, поскольку они могут отключить исполнение логики в методе *insert*, отправку оповещений, а также отключить запись журнала БД.

Если вы перекрываете метод *insert*, то воспользуйтесь перекрестными ссылками, чтобы определить, вызывается ли *skipDataMethods(true)* где-либо в коде X++. Если вы этого не сделаете, то код X++ может в результате не вызвать перекрытый метод *insert*. Более того, если вы вызываете в своем коде *skipDataMethods(true)*, убедитесь, что из-за пропуска выполнения перекрытого метода *insert* данные не окажутся в несогласованном состоянии.

Вызовы *skip*-методов лишь оказывают влияние на то, будет ли понижен уровень оператора *insert_recordset* до работы с отдельными записями. Если вы вызываете *skipDataMethods(true)* для предотвращения такого понижения, потому что на таблице перекрыт метод *insert*, воспользуйтесь Синтаксическим анализатором Microsoft Dynamics AX (Trace Parser), чтобы убедиться, что такого понижения на самом деле не произошло.

Операция понижается до уровня работы с отдельными записями, если, к примеру, на вставку записей в таблицу настроено логирование в журнал БД. В предыдущем примере перекрытый метод *insert* на таблице *InsertInventTableInventSum* выполнялся бы, если бы на вставку записей в *InsertInventTableInventSum* было настроено логирование в журнал БД, потому что оператор *insert_recordset* был бы преобразован в цикл *while select*, внутри которого выполнялся перекрытый метод *insert*. Более подробную информацию о Синтаксическом анализаторе вы можете найти в разделе «Средства мониторинга производительности» далее в этой главе.

С версии Microsoft Dynamics AX 2009 оператор *insert_recordset* поддерживает литералы. Их поддержка была введена главным образом для использования в сценариях обновления, когда целевая таблица заполняется записями из одной или нескольких исходных таблиц, объединенных в запросе, а одно или несколько полей в целевой таблице должны быть заполнены значением литерала, которое не существует в исходных данных. Следующий пример кода иллюстрирует использование литералов в *insert_recordset*.

```
static void CopyItemInfoLiteralSample(Args _args)
{
    InventTable                 inventTable;
    InventSum                   inventSum;
    InsertInventTableInventSum  insertInventTableInventSum;
    boolean                     flag = boolean::true;

    ttsBegin;
    insert_recordset insertInventTableInventSum
(ItemId,AltItemId,PhysicalValue,PostedValue,Flag)
        select ItemId,altitemid from inventTable where inventTable.ItemId == '1001'
        join PhysicalValue,PostedValue,flag from inventSum
        where inventSum.ItemId == inventTable.ItemId;
    ttsCommit;

    select firstonly ItemId,Flag from insertInventTableInventSum;
    info(strFmt('%1,%2',insertInventTableInventSum.ItemId,insertInventTableInventSum.
Flag));
    // далее следует код, использующий скопированные данные...
}
```

Оператор *update_recordset*

Поведение оператора *update_recordset* очень похоже на поведение оператора *insert_recordset*. Это иллюстрируется в следующем примере кода, в котором все записи, которые были вставлены для одного кода номенклатуры, обновляются и помечаются флагом для последующей обработки.

```
static void UpdateCopiedData(Args _args)
{
    InventTable                 inventTable;
    InventSum                   inventSum;
    InsertInventTableInventSum  insertInventTableInventSum;

    // Код предполагает, что InsertInventTableInventSum уже заполнена.
    // Операция обновления на основе наборов.
    ttsBegin;
    update_recordSet insertInventTableInventSum setting Flag = true
    where insertInventTableInventSum.ItemId == '1001';
    ttsCommit;
}
```

Исполнение оператора *update_recordset* приводит к передаче одного оператора базе данных, который в Transact-SQL использует следующий синтаксис: *UPDATE <таблица> <SET> <список полей и выражений>*

WHERE <предикаты>. Как и в случае оператора *insert_recordset*, данный оператор обеспечивает гигантское повышение производительности по сравнению с версией на основе записей, в которой каждая запись обновляется индивидуально. Это демонстрируется в следующем коде X++, который выполняет те же действия, что и в предыдущем примере. В коде производится выбор записей, удовлетворяющих условию обновления, устанавливается новое значение поля описания и вызывается метод обновления записи.

```
static void UpdateCopiedDataLineBased(Args _args)
{

    InventTable                 inventTable;
    InventSum                   inventSum;
    InsertInventTableInventSum  insertInventTableInventSum;

    // Код предполагает, что InsertInventTableInventSum уже заполнена

    ttsBegin;
    while select forUpdate InsertInventTableInventSum
    where insertInventTableInventSum.ItemId == '1001'
   {
        insertInventTableInventSum.Flag = true;
        insertInventTableInventSum.update();
   }
    ttsCommit;
}
```

Если условию обновления удовлетворяют 10 записей, то на исполнение базе данных будут переданы один оператор *select* и 10 операторов *update* вместо отправки единственного оператора *update* при использовании *update_recordset*.

Оператор *update_recordset* также может быть понижен, если перекрыты определенные методы или Microsoft Dynamics AX настроена определенным образом. Это происходит при выполнении любого из следующих условий.

- Таблица кэшируется с использованием настройки *EntireTable*.
- На целевой таблице перекрыт метод *update*, *aosValidateUpdate* или *aosValidateRead*.
- На обновление записей целевой таблицы настроено срабатывание оповещений.

- На обновление записей целевой таблицы настроено логирование в журнал БД.
- На целевой таблице настроена безопасность на уровне записей (RLS).
- Свойство таблицы *ValidTimeStateFieldType* установлено в значение, отличное от *None*.

Среда времени выполнения Microsoft Dynamics AX автоматически обрабатывает понижение и внутренне исполняет код, который похож на сценарий из предыдущего примера, использующий оператор *while select*.

Как и с оператором *insert_recordset*, избежать понижения уровня операции можно вопреки любой упомянутой выше функциональности, кроме случая, когда кэшируется вся таблица полностью. Буфер записи содержит методы, отключающие проверки, которые выполняет среда времени выполнения для определения, следует ли понизить оператор *update_recordset* до оператора на уровне записей.

- Вызов *skipDataMethods(true)* отключает проверку, определяющую, был ли перекрыт метод *update*.
- Вызов *skipAosValidation(true)* отключает проверки в методах *aosValidateUpdate* и *aosValidateRead*.
- Вызов *skipDatabaseLog(true)* отключает проверку, определяющую, настроено ли логирование в журнал БД обновления записей таблицы.
- Вызов *skipEvents(true)* отключает проверку, определяющую, настроены ли на обновление записей таблицы какие-либо уведомления.

Как указывалось ранее, *skip*-методы следует использовать с осторожностью. Опять-таки использование *skip*-методов лишь оказывает влияние на то, будет ли понижен оператор *update_recordset* до операции *while select*. Если понижение все равно происходит, то логирование в журнал БД, отправка оповещений и выполнение перекрытых методов выполняются несмотря на то, что были вызваны соответствующие *skip*-методы.

Совет. Если оператор *update_recordset* понижается до уровня оператора *while select*, то оператор *select* использует конкурентную модель, указанную на уровне таблицы. Однако можно указать ключевые слова *optimisticlock* или *pessimisticlock* на операторе *update_recordset*, что обеспечит применение указанной конкурентной модели в случае понижения уровня оператора.

Microsoft Dynamics AX поддерживает объединение таблиц типа *inner join* и *outer join* в *update_recordset*. Благодаря этому оператор *update_recordset* позволяет выполнять операции на основе набора в случаях, когда исходные данные выбираются из более чем одной связанной таблицы.

Следующий пример иллюстрирует использование объединений таблиц в *update_recordset*.

```
static void UpdateCopiedDataJoin(Args _args)
{
    InventTable                 inventTable;
    InventSum                   inventSum;
    InsertInventTableInventSum  insertInventTableInventSum;

    // Код предполагает, что InsertInventTableInventSum уже заполнена
    // Операция обновления на основе набора с использованием объединений таблиц

    ttsBegin;
    update_recordSet insertInventTableInventSum setting Flag = true,
    DiffAvailOrderedPhysical = inventSum.AvailOrdered - inventSum.AvailPhysical
    join InventSum where inventSum.ItemId == insertInventTableInventSum.ItemId &&
    inventSum.AvailOrdered > inventSum.AvailPhysical;
    ttsCommit;
}
```

Оператор *delete_from*

Оператор *delete_from* схож с операторами *insert_recordset* и *update_recordset* в том, что он передает базе данных единственный оператор для удаления одновременно нескольких записей, как показано в следующем примере.

```
static void DeleteCopiedData(Args _args)
{
    InventTable                 inventTable;
    InventSum                   inventSum;
    InsertInventTableInventSum  insertInventTableInventSum;

    // Код предполагает, что InsertInventTableInventSum уже заполнена
    // Операция удаления на основе набора

    ttsBegin;
    delete_from insertInventTableInventSum
    where insertInventTableInventSum.ItemId == '1001';
    ttsCommit;
}
```

Этот код передает в Microsoft SQL Server оператор, по синтаксису схожий с *DELETE <таблица> WHERE<предикаты>*, и выполняет те же действия, что и следующий код X++, удаляющий по одной записи.

```
static void DeleteCopiedDataLineBased(Args _args)
{
    InventTable                 inventTable;
    InventSum                   inventSum;
    InsertInventTableInventSum  insertInventTableInventSum;

    // Код предполагает, что InsertInventTableInventSum уже заполнена

    ttsBegin;
    while select forUpdate insertInventTableInventSum
    where insertInventTableInventSum.ItemId == '1001'
    {
        insertInventTableInventSum.delete();
    }
    ttsCommit;
}
```

Опять же использование оператора *delete_from* предпочтительнее в отношении производительности, поскольку базе данных передается один оператор вместо нескольких, как в случае использования операторов на основе записей.

Подобно тому, как могут быть понижены операторы *insert_recordset* и *update_recordset*, может быть понижен и оператор *delete_from*. Понижение происходит при выполнении любого из следующих условий.

- Таблица кэшируется с использованием настройки *EntireTable*.
- На целевой таблице перекрыт метод *delete*, *aosValidateDelete* или *aosValidateRead*.
- На удаление записей целевой таблицы настроено срабатывание оповещений.
- На удаление записей целевой таблицы настроено логирование в журнал БД.
- Свойство таблицы *ValidTimeStateFieldType* установлено в значение, отличное от *None*.

Также понижение происходит в том случае, если на таблице определены действия при удалении (Delete Actions). Среда времени выполнения

Microsoft Dynamics AX автоматически обрабатывает понижение и внутренне исполняет код, который похож на сценарий из предыдущего примера, использующий *while select*.

Избежать понижения уровня операции можно вопреки любой упомянутой выше функциональности, кроме случая, когда кэшируется вся таблица полностью. Буфер записи содержит методы, отключающие проверки, которые выполняет среда времени выполнения для определения, следует ли понизить оператор *delete_recordset* до оператора на уровне записей.

- Вызов *skipDataMethods(true)* отключает проверку, определяющую, был ли перекрыт метод *delete*.
- Вызов *skipAosValidation(true)* отключает проверки в методах *aosValidateDelete* и *aosValidateRead*.
- Вызов *skipDatabaseLog(true)* отключает проверку, определяющую, настроено ли логирование в журнал БД удаления записей таблицы.
- Вызов *skipEvents(true)* отключает проверку, определяющую, настроены ли на удаление записей таблицы какие-либо уведомления.

Предыдущие замечания для оператора *update_recordset*, касающиеся использования *skip*-методов, игнорирование пропуска действий в случае понижения и используемой конкурентной модели в равной степени относятся и к оператору *delete_recordset*.

Примечание. На табличном буфере есть метод *skipDeleteMethod*. Вызов *skipDeleteMethod(true)* имеет тот же эффект, что и вызов метода *skipDataMethods(true)*, он задействует ту же логику среды времени выполнения Microsoft Dynamics AX, поэтому вы можете использовать *skipDeleteMethod* в комбинации с *insert_recordset* и *update_recordset*, хотя это не лучшим образом скажется на читаемости кода X++.

Классы *RecordInsertList* и *RecordSortedList*

В дополнение к операторам на основе наборов, при вставке нескольких записей в таблицы Microsoft Dynamics AX также позволяет использовать классы *RecordInsertList* и *RecordSortedList*. Когда записи готовы к вставке в базу данных, среда времени выполнения Microsoft Dynamics AX упаковывает несколько записей в один пакет и отправляет его на сервер базы

данных, после чего база данных выполняет индивидуальную вставку каждой записи в пакете. Это иллюстрируется в следующем примере, в котором создается объект класса *RecordInsertList*, затем каждая запись, которая должна быть вставлена в базу данных, добавляется к объекту этого класса. Когда все требуемые записи уже добавлены к объекту, вызывается метод *insertDatabase*, который вставляет в базу данных все оставшиеся записи.

```
static void CopyItemInfoRIL(Args _args)
{
    InventTable                     inventTable;
    InventSum                       inventSum;
    InsertInventTableInventSumRT    insertInventTableInventSumRT;
    RecordInsertList ril;

    ttsBegin;
    ril = new RecordInsertList(tableNum(InsertInventTableInventSumRT));

    while select ItemId,AltItemid from inventTable where inventTable.ItemId == '1001'
    join PhysicalValue,PostedValue from inventSum
    where inventSum.ItemId == inventTable.ItemId
    {
        insertInventTableInventSumRT.ItemId          = inventTable.ItemId;
        insertInventTableInventSumRT.AltItemId       = inventTable.AltItemId;
        insertInventTableInventSumRT.PhysicalValue   = inventSum.PhysicalValue;
        insertInventTableInventSumRT.PostedValue     = inventSum.PostedValue;
        // Если пакет наполнен, происходит вставка записей
        ril.add(insertInventTableInventSumRT);
    }

    // Вставка оставшихся записей в БД

    ril.insertDatabase();
    ttsCommit;

    select count(RecId) from insertInventTableInventSumRT;
    info(int642str(insertInventTableInventSumRT.RecId));

    // Дополнительный код, использующий созданные записи...
}
```

В зависимости от параметра размера буфера *Maximum buffer size* в настройках сервера приложений среда времени выполнения Microsoft Dynamics AX определяет количество записей, которое поместится в бу-

фер, как функцию от размера одной записи и максимального размера буфера. Если среда времени выполнения Microsoft Dynamics AX обнаруживает, что достаточное для формирования пакета количество записей было добавлено в объект *RecordInsertList*, то записи упаковываются, передаются на сервер базы данных и вставляются в индивидуальном порядке в требуемую таблицу. Данная проверка выполняется при вызове метода *add*. Когда из прикладной логики вызывается метод *insertDatabase*, то с использованием того же механизма вставляются оставшиеся записи.

Использование этих классов вместо сценария с использованием оператора *while select* дает следующее преимущество: выполняется меньшее количество обращений AOS к базе данных, поскольку одновременно отправляются несколько записей. Однако число операторов *INSERT* на уровне базе данных остается таким же.

Примечание. Поскольку время вставки записей в базу данных зависит от размера буфера записей и размера пакета, не стоит ожидать, что запись можно будет выбрать из базы данных раньше, чем произойдет вызов метода *insertDatabase*.

Предыдущий сценарий может быть переписан с использованием класса *RecordSortedList* вместо класса *RecordInsertList*, как показано в коде X++ ниже.

```
public static server void CopyItemInfoRSL()
{
    InventTable                 inventTable;
    InventSum                   inventSum;
    InsertInventTableInventSumRT insertInventTableInventSumRT;
    RecordSortedList rsl;

    ttsBegin;
    rsl = new RecordSortedList(tableNum(InsertInventTableInventSumRT));
    rsl.sortOrder(fieldNum(InsertInventTableInventSumRT,PostedValue));

    while select ItemId,AltItemid from inventTable where inventTable.ItemId == '1001'
    join PhysicalValue,PostedValue from inventSum
    where inventSum.ItemId == inventTable.ItemId
    {
        insertInventTableInventSumRT.ItemId          = inventTable.itemId;
        insertInventTableInventSumRT.AltItemId       = inventTable.AltItemId;
        insertInventTableInventSumRT.PhysicalValue   = inventSum.PhysicalValue;
```

```
        insertInventTableInventSumRT.PostedValue     = inventSum.PostedValue;

        // записи в БД не вставляются
        rsl.ins(insertInventTableInventSumRT);
    }

    // все записи вставляются в БД
    rsl.insertDatabase();
    ttsCommit;

    select count(RecId) from insertInventTableInventSumRT;
    info(int642str(insertInventTableInventSumRT.RecId));

    // Дополнительный код, использующий созданные записи...
}
```

Когда прикладная логика использует объект класса *RecordSortedList*, записи не отправляются для вставки в базу данных до тех пор, пока не будет вызван метод *insertDatabase*. Количество обращений и исполняемых операторов *INSERT* на уровне базы данных такое же, как и у объекта класса *RecordInsertList*.

И RecordInsertList, и *RecordSortedList* могут быть понижены средой времени выполнения до уровня вставки на основе записей, когда каждая запись отправляется базе данных в отдельном цикле приема-передачи, после чего выполняется отдельный оператор *INSERT*. Это происходит, если перекрыт метод *insert* или метод *aosValidateInsert* или же если таблица содержит поля типа *container* или *memo*. Однако понижение не происходит, если на вставку в целевую таблицу настроена запись в журнал базы данных или отправка оповещений. Исключением из этого правила является случай, когда настроены протоколирование или оповещения и при этом в таблице есть поля *CreatedDateTime* или *ModifiedDateTime* – в этом случае происходит вставка на основе записей. Протоколирование базы данных и генерация оповещений происходит последовательно, запись за записью, уже после того, как записи были отправлены и вставлены в базу данных.

При создании объекта класса *RecordInsertList* можно указать, что методы *insert* и *aosValidateInsert* должны пропускаться. Также можно указать, что если не происходит понижение операции, то протоколирование базы данных и генерация оповещений должны быть проигнорированы.

Советы по преобразованию кода для использования операций на основе наборов

Зачастую код не преобразуется для использования операций на основе наборов из-за того, что его логика слишком сложна. Однако, к примеру, условие *if* может быть размещено в выражении *where* запроса. Если в вашем случае требуется принятие решений с использованием ветвления *if/else*, то это можно реализовать с помощью двух запросов, таких как два оператора *update_recordset*. Необходимая информация из других таблиц может быть получена с использованием объединения таблиц в запросе, вместо того чтобы выбирать соответствующие записи с помощью *find*. В Microsoft Dynamics AX 2012 *insert_recordset* и временные таблицы типа *TempDB* помогают расширить возможности преобразования кода для использования операций на основе наборов.

Однако некоторые возможности, такие как вычисления над полями в операторе *select*, могут казаться трудно реализуемыми при использовании операций на основе наборов. По этой причине Microsoft Dynamics AX 2012 предлагает для представлений возможность, называемую *вычисляемые поля*, и вы можете использовать ее для преобразования достаточно сложной логики в операции на основе наборов. Вычисляемые поля также могут предоставить преимущество в плане производительности, когда они используются в качестве альтернативы дисплейным методам на источниках данных, доступных только для чтения. Вообразите себе такую задачу: найти всех клиентов, которые приобрели товаров больше чем на $100000, и всех клиентов, которые приобрели товаров больше чем на $1000000. Такие клиенты рассматриваются как VIP-клиенты, которые затем получают определенные скидки.

В более ранних версиях Microsoft Dynamics AX код X++ для установки соответствующих признаков у клиентов выглядел бы примерно так, как показано в следующем примере.

```
public static server void demoOld()
{
    SalesLine   sl;
    CustTable   ct;
    vipparm     vp;
    int64       total;

    vp = vipparm::find();
    ttsBegin;
```

```
    // Один + n циклов приема-передачи на одного клиента в таблице SalesLine
    while select CustAccount, sum(SalesQty), sum(SalesPrice) from sl group by sl.CustAccount
    {

    // Необходимо выбрать запись на обновление, что приводит
    // к n дополнительных циклов приема-передачи
    ct = CustTable::find(sl.CustAccount,true);
    ct.VIPStatus = 0;

        if ((sl.SalesQty*sl.SalesPrice)>=vp.UltimateVIP)
            ct.VIPStatus = 2;
        else if ((sl.SalesQty*sl.SalesPrice)>=vp.VIP)
            ct.VIPStatus = 1;

    // Еще n циклов приема-передачи для обновления
        if (ct.VIPStatus != 0)
            ct.update();
    }
    ttsCommit;
}
```

Чтобы повысить эффективность такого обновления, вы могли бы заменить его двумя прямыми запросами Transact-SQL. Эти запросы выглядели бы примерно так:

```
UPDATE CUSTTABLE SET VIPSTATUS = 2 FROM (SELECT CUSTACCOUNT,SUM(SALESQTY)*S
UM(SALESPRICE) AS TOTAL,
VIPSSTATUS = CASE
    WHEN SUM(SALESQTY)*SUM(SALESPRICE) > 1000000 THEN 2
    WHEN SUM(SALESQTY)*SUM(SALESPRICE) > 100000 THEN 1
    ELSE 0 END
    FROM SALESLINE GROUP BY CUSTACCOUNT) AS VC
WHERE VC.VIPSTATUS = 2 and CUSTTABLE.ACCOUNTNUM = VC.CUSTACCOUNT and
DATAAREAID = N'CEU'
```

Примечание. Этот код содержит лишь часть условий по *dataAreaId* и ни одного – по *Partition*, что подчеркивает его слабости. Логика доступа к данным тут не применяется в должной мере.

В Microsoft Dynamics AX 2012 с помощью вычисляемых полей вы можете заменить этот код с помощью двух операторов на основе наборов. Чтобы написать эти операторы, вам сначала необходимо создать объект Query в AOT, потому что представления сами по себе не могут содержать

оператор *group by*. Далее вам понадобится параметрическая таблица, в которой будет содержаться информация о том, кто считается VIP-клиентом в каждой компании (рис. 13-4). Затем вам нужно объединить всю эту информацию, чтобы она была доступна во время выполнения. Код **вычисляемого** поля показан ниже.

```
private static server str compColQtyPrice()
{
    str sReturn,sQty,sPrice,ultimateVIP,VIP;
    Map m = new Map(Types::String,Types::String);
    sQty        = SysComputedColumn::returnField(tableStr(mySalesLineView),
                                                 identifierStr(SalesLine_1),
                                                 fieldStr(SalesLine,SalesQty));
    sPrice      = SysComputedColumn::returnField(tableStr(mySalesLineView),
                                                 identifierStr(SalesLine_1),
                                                 fieldStr(SalesLine,SalesPrice));
    ultimateVIP = SysComputedColumn::returnField(tableStr(mySalesLineView),
                                                 identifierStr(Vipparm_1),
                                                 fieldStr(vipparm,ultimateVIP));
    VIP         = SysComputedColumn::returnField(tableStr(mySalesLineView),
                                                 identifierStr(Vipparm_1),
                                                 fieldStr(vipparm,VIP));
    m.insert(SysComputedColumn::sum(sQty)+'*'+SysComputedColumn::sum(sPrice)+
             ' > '+ultimateVIP,int2str(VipStatus::UltimateVIP));
    m.insert(SysComputedColumn::sum(sQty)+'*'+SysComputedColumn::sum(sPrice)+
             ' > '+VIP,int2str(VipStatus::VIP));
    return SysComputedColumn::switch('',m,'0');
}
```

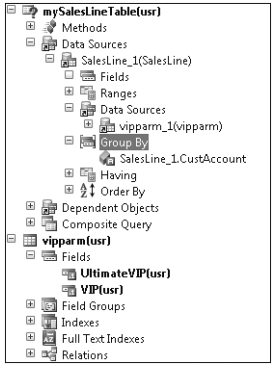

Рис. 13–4. Создание параметрической таблицы и исходного запроса

Следующим шагом нужно добавить в представление параметрическую таблицу и создать необходимое вычисляемое поле, как показано на рис. 13-5.

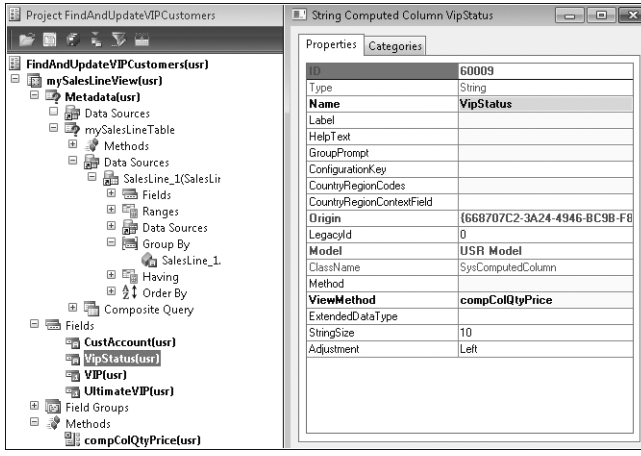

Рис. 13–5. Создание представления и вычисляемого поля

В SQL Server представление будет выглядеть следующим образом:

```
SELECT T1.CUSTACCOUNT AS CUSTACCOUNT,T1.DATAAREAID AS DATAAREAID,1010
AS RECID,T2.DATAAREAID AS DATAAREAID#2,T2.VIP AS VIP,T2.ULTIMATEVIP AS
ULTIMATEVIP,(CAST ((CASE WHEN SUM(T1.SALESQTY)*SUM(T1.SALESPRICE) >
T2.ULTIMATEVIP THEN 2 WHEN SUM(T1.SALESQTY)*SUM(T1.SALESPRICE) > T2.VIP THEN 1
ELSE 0 END) AS NVARCHAR(10))) AS VIPSTATUS FROM SALESLINE T1 CROSS JOIN VIPPARM
T2 GROUP BY T1.CUSTACCOUNT,T1.DATAAREAID,T2.DATAAREAID,T2.VIP,T2.ULTIMATEVIP
```

Теперь вы можете заменить используемый ранее код обновления на основе отдельных записей более эффективным кодом, работающим на основе наборов.

```
public static server void demoNew()
{
    mySalesLineView    mySLV;
    CustTable          ct;
    ct.skipDataMethods(true);
    update_recordSet ct setting VipStatus = VipStatus::UltimateVIP
    join mySLV where ct.AccountNum == mySLV.CustAccount &&
    mySLV.VipStatus == int2str(enum2int(vipstatus::UltimateVIP));
    update_recordSet ct setting VipStatus = VipStatus::VIP
    join mySLV where ct.AccountNum == mySLV.CustAccount &&
    mySLV.VipStatus == int2str(enum2int(vipstatus::VIP));
}
```

Запустим код, чтобы продемонстрировать разницу в затрачиваемом на выполнение времени.

```
public static void main(Args _args)
{
    int tickcnt;
    DemoClass::resetCusttable();
    tickcnt = WinAPI::getTickCount();
    DemoClass::demoOld();
    info('На основе отдельных записей ' + int2str(WinAPI::getTickCount()-tickcnt));
    DemoClass::resetCusttable();
    tickcnt = WinAPI::getTickCount();
    DemoClass::demoNew();
    info('На основе наборов записей ' + int2str(WinAPI::getTickCount()-tickcnt));
}
```

Время выполнения операций получилось следующим:

- **на основе отдельных записей –** 1514 мс;
- **на основе наборов записей –** 171 мс.

Заметьте, что мы запускали код на демо-данных; вообразите, какова была бы разница на реальной базе данных с сотнями тысяч заказов на продажу и клиентов.

Другой пример, который было бы сложно преобразовать для использования операции на основе наборов, – когда вам нужно использовать в запросах агрегирование и *group by*, поскольку оператор *update_recordset* это не поддерживает. Обойти ограничение вы можете с помощью временных таблиц типа *TempDB*, а также комбинации *insert_recordset* и *update_recordset*.

Примечание. Целесообразность применения этого подхода определяется объемом данных, которые вы хотите обновить. К примеру, если вы хотите обновить 10 записей, то использование *while select* будет более эффективным. Но если вам нужно обновить сотни и тысячи записей, то более эффективным будет предложенный тут подход. В вашем конкретном случае нужно провести оценку производительности и проверить каждый подход по отдельности, чтобы определить, какой обеспечит более высокую производительность.

В следующем примере сначала заполняется таблица, а потом значения в ней обновляются в зависимости от результатов операций *group by* и *sum* в выражении. Следует заметить, что удаление и заполнение данных занимает больше времени, чем последующее выполнение операторов *insert_recordset* и *update_recordset*.

```
public static server void PopulateTable()
{
    MyUpdRecordsetTestTable MyUpdRecordsetTestTable;
    int myGrouping,myKey,mySum;
    RecordInsertList ril = new RecordInsertList(tablenum(MyUpdRecordsetTestTable));

    delete_from MyUpdRecordsetTestTable;

    for(myKey=0;myKey<=100000;myKey++)
    {
        MyUpdRecordsetTestTable.Key              = myKey;
        if(myKey mod 10 == 0)
        {
            myGrouping += 10;
            mySum += 10;
        }
        MyUpdRecordsetTestTable.fieldForGrouping          = myGrouping;
        MyUpdRecordsetTestTable.theSum                    = mySum;
        ril.add(MyUpdRecordsetTestTable);
    }
    ril.insertDatabase();
}
```

Сочетание вместе временных таблиц типа *TempDB*, операторов *insert_recordset* и *update_recordset* для обновления записей в таблице:

```
public static void InsertAndUpdate()
{
    MyUpdRecordsetTestTable MyUpdRecordsetTestTable;
    MyUpdRecordsetTestTableTmp MyUpdRecordsetTestTableTmp;
    int tc;

    tc = WinAPI::getTickCount();
    insert_recordset MyUpdRecordsetTestTableTmp(fieldForGrouping,theSum)
    select fieldForGrouping,sum(theSum) from MyUpdRecordsetTestTable
    Group by MyUpdRecordsetTestTable.fieldForGrouping;
    info("Time needed: " + int2str(WinAPI::getTickCount()-tc));
```

```
    tc = WinAPI::getTickCount();
    update_recordSet MyUpdRecordsetTestTable setting theSum =
MyUpdRecordsetTestTableTmp.theSum
    join MyUpdRecordsetTestTableTmp
    where MyUpdRecordsetTestTable.fieldForGrouping == MyUpdRecordsetTestTableTmp.
fieldForGrouping;
    info("Time needed: " + int2str(WinAPI::getTickCount()-tc));
}
```

При запуске этого кода на демо-данных время выполнения операторов получилось таким:

- **выражение с *insert_recordset*** – 1685 мс;
- **выражение с *update_recordset*** – 3697 мс.

Перезапускаемые задания и оптимистичная конкурентная модель

В Microsoft Dynamics AX во многих случаях выполнение определенной прикладной логики включает в себя манипулирование одновременно несколькими строками в одной и той же таблице. Некоторые сценарии требуют, чтобы манипуляции над всеми строками выполнялись в рамках одной транзакции; если что-то пойдет не так и транзакция отменяется, то все изменения в базе данных откатываются, а задание может быть перезапущено вручную или автоматически. Другие сценарии фиксируют изменения в базе данных после изменения каждой из записей; в случае сбоя откатываются только изменения текущей записи, а все ранее обработанные записи уже зафиксированы. Когда задание перезапускается в таком сценарии, оно начинается с точки сбоя, пропуская все записи, которые уже были успешно изменены.

Пример первого сценария представлен в коде ниже, в котором все вызовы *update* на записях таблицы *CustTable* выполняются в одной транзакции базы данных.

```
static void UpdateCreditMax(Args _args)
{
    CustTable custTable;

    ttsBegin;
    while select forupdate custTable where custTable.CreditMax == 0
    {
        if (custTable.balanceMST() < 10000)
```

```
            {
                custTable.CreditMax = 50000;
                custTable.update();
            }
        }
        ttsCommit;
    }
```

Пример второго сценария, исполняющего ту же самую логику, приводится в следующем коде, в котором каждая запись обрабатывается в отдельной транзакции. Обратите внимание на то, что вы должны заново выбирать для обновления каждую запись таблицы *CustTable* внутри транзакции для того, чтобы среда времени выполнения Microsoft Dynamics AX позволила обновить запись.

```
static void UpdateCreditMax(Args _args)
{
    CustTable  custTable;
    CustTable  updateableCustTable;
    while select custTable where custTable.CreditMax == 0
    {
        if (custTable.balanceMST() < 10000)
        {
            ttsBegin;
            select forupdate updateableCustTable
                where updateableCustTable.AccountNum == custTable.AccountNum;

            updateableCustTable.CreditMax = 50000;
            updateableCustTable.update();
            ttsCommit;
        }
    }
}
```

В сценарии, где 100 записей таблицы *CustTable* удовлетворяют условию обновления, в первом примере базе данных отравляются один оператор *select* и 100 операторов *update*, а второй пример использует один большой запрос *select* 100 небольших запросов *select* плюс еще 100 операторов *update*. Поэтому первый сценарий будет выполняться быстрее, чем второй, но при этом первый сценарий будет также удерживать блокировку обновляемых в таблице *CustTable* записей более длительное время, поскольку он не фиксирует каждую запись в отдельности. Второй пример демонстрирует более удачный подход с точки зрения параллельной обработки данных

по сравнению с первым примером, поскольку блокировки удерживаются в течение чрезвычайно малого промежутка времени.

Оптимистичная конкурентная модель в Microsoft Dynamics AX позволяет воспользоваться преимуществами обоих вышеописанных примеров. Вы можете выбрать записи за пределами блока транзакции и обновить их внутри области действия транзакции, но только если записи были выбраны с использованием оптимистичной конкурентной модели. Это демонстрируется в следующем примере, в котором к оператору *select* применяется ключевое слово *optimisticklock* и каждая запись обновляется в отдельной транзакции базы данных. Поскольку записи выбираются с использованием ключевого слова *optimisticklock*, нет необходимости в повторном индивидуальном выборе на обновление каждой записи внутри области действия транзакции.

```
static void UpdateCreditMax(Args _args)
{
    CustTable custTable;

    while select optimisticlock custTable where custTable.CreditMax == 0
    {
        if (custTable.balanceMST() < 10000)
        {
            ttsBegin;
            custTable.CreditMax = 50000;
            custTable.update();
            ttsCommit;
        }
    }
}
```

Данный подход использует то же количество обращений к серверу базы данных, что и в первом примере, при этом улучшая конкуретность благодаря фиксации данных в базе для каждой записи в отдельности. Но данный пример все же не выполняется так же быстро, как первый, поскольку у него есть накладные расходы на управление транзакцией для каждой записи. Вы можете продолжить дальнейшую оптимизацию производительности данного примера, манипулируя числом записей, для которых открывается отдельная транзакция. Однако соответствующий выбор частоты фиксации изменений всегда зависит от конкретных обстоятельств, в которых выполняется задание.

Совет. Вы можете использовать ключевое слово *forupdate*, когда производится выбор записей за пределами транзакции, если оптимистичная конкурентная модель задана на уровне свойств таблицы. Однако рекомендуется явно использовать ключевое слово *optimisticklock*, поскольку сценарий будет продолжать корректно работать, даже если настройка конкурентной модели на уровне таблицы будет изменена. Использование ключевого слова *optimisticklock* также улучшает читаемость кода X++, явно указывая намерения разработчика использовать оптимистичную конкурентную модель.

Кэширование

Среда времени выполнения Microsoft Dynamics AX поддерживает кэширование на основе одиночных записей и на основе набора записей. Кэширование на основе набора может настраиваться в метаданных путем переключения свойства в определении таблицы или путем явного написания кода X++, который создает кэш. Вне зависимости от того, как настроен кэш, вам не нужно знать, какой метод кэширования используется в конкретном случае, поскольку среда времени выполнения обрабатывает кэш без вмешательства разработчика. Но для оптимизации использования кэша необходимо понимать, как работает каждый из механизмов кэширования.

В Microsoft Dynamics AX 2012 появились некоторые важные возможности. К примеру, кэширование на основе записей работает не только для отдельной записи, но также и для объединений в запросах. Этот механизм описан в следующем разделе, «Кэширование одиночных записей». Кроме того, даже если в запросе используются предикаты, сужающие выборку, кэширование будет продолжать работать при условии, что в запросе содержится выборка по полям уникального ключа.

Руководство разработчика Microsoft Dynamics AX 2012 (SDK) содержит хорошее описание отдельных возможностей кэширования и того, как они настраиваются (см. раздел «Record Caching» по адресу: *http://msdn.microsoft.com/en-us/library/bb278240.aspx)*.

В этом разделе основное внимание уделяется тому, как кэши реализованы в среде времени выполнения Microsoft Dynamics AX и какого по-

ведения системы следует ожидать при использовании определенных механизмов кэширования.

Кэширование одиночных записей

На таблице можно настроить три типа кэширования на основе одиночных записей, для свойства *CacheLookup* в определении таблицы нужно установить в одно из следующих значений: *Found, FoundAndEmpty* или *NotInTTS*. Дополнительное значение (помимо *None*) – это *EntireTable*, настройка для кэширования на основе набора записей. Эти настройки вкратце были описаны в разделе «Кэширование и индексация» ранее в этой главе, здесь они обсуждаются более детально.

Возможности трех типов кэширования на основе записей по сути своей одни и те же. Разница заключается только в том, что кэшируется и когда кэшированные значения очищаются. Например кэш, использующий настройку *Found* или *FoundAndEmpty*, сохраняется при входе/выходе из области действия транзакции базы данных. Таблица, использующая кэш *NotInTTS,* не использует его при первом доступе к таблице внутри области действия транзакции – при этом кэш используется во всех последующих операторах *select*, кроме случая, когда к запросу применяется ключевое слово *forupdate*.

В следующем примере кода X++ показывается, когда кэш будет использоваться внутри и за пределами области действия транзакции, если таблица использует механизм кэширования *NotInTTS*, а поле *AccountNum* является первичным ключом таблицы. Комментарии в коде описывают, когда кэш будет использоваться, а когда нет. В примере показано, что первые два оператора *select* после команды *ttsbegin* не будут использовать кэш. Первый оператор не будет использовать кэш, поскольку он является первым оператором внутри области действия транзакции, а второй оператор – поскольку к нему применено ключевое слово *forupdate*. Использование этого ключевого слова заставляет среду времени выполнения осуществлять поиск записи в базе данных, поскольку кэшированная версия записи не была выбрана для обновления при помощи ключевого слова *forupdate*.

```
static void NotInTTSCache(Args _args)
{
    CustTable custTable;

    select custTable                                // Поиск записи в кэше. Если она не
        where custTable.AccountNum == '1101'; //найдена, то поиск в БД
```

```
    ttsBegin;                                   // Начало транзакции

    select custTable                            // Кэш недействителен.
        where custTable.AccountNum == '1101';   // Искать в БД и поместить в кэш

    select forupdate custTable                  // Искать в БД, потому что
        where custTable.AccountNum == '1101';   // используется ключевое слово forupdate.

    select custTable                            // Будет использован кэш, из БД ничего.
        where custTable.AccountNum == '1101';   // не выбирается.

    select forupdate custTable                  // Будет использован кэш, потому что
        where custTable.AccountNum == '1101';   // лючевое слово forupdate уже было
                                                // к использовано ранее.

    ttsCommit;                                  // Конец транзакции.

    select custTable                            // Будет использован кэш.
        where custTable.AccountNum == '1101';
}
```

Если бы таблица в предыдущем примере была настроена на использование кэширования типа *Found* или *FoundAndEmpty*, то при исполнении первого оператора *select* внутри транзакции кэш бы использовался, а при исполнении первого оператора *select*, к которому применено ключевое слово *forupdate*, – нет.

Примечание. По умолчанию все системные таблицы Microsoft Dynamics AX настроены на использование кэширования типа *Found*. Изменить эту настройку нельзя.

Для всех трех механизмов кэширования кэш используется только тогда, когда оператор *select* содержит предикаты равенства (==) в условии *where*, и при этом все поля в точности совпадают с полями в первичном или любом другом уникальном индексе таблицы. Следовательно, название одного из уникальных индексов, которые используются для доступа к кэшу из прикладной логики, должно быть указано в свойстве *PrimaryIndex* таблицы. Для любого другого уникального индекса, не требуя дополнительных установок в метаданных таблицы, ядро приложения будет автоматически использовать кэш, если соответствующая запись в нем уже присутствует.

Следующие примеры кода X++ показывают, когда среда времени выполнения Microsoft Dynamics AX будет пытаться использовать кэш, а когда нет. Кэш в следующем примере будет использоваться только в первом и последнем операторе *select*. Другие два оператора будут выполнять поиск записи в базе данных.

```
static void UtilizeCache(Args _args)
{
    CustTable custTable;

    select custTable                              // Будет использован кэш, потому что в
        where custTable.AccountNum == '1101';     // условии where используется
                                                  // первичный ключ

    select custTable;                             // Не может использовать кэш, потому
                                                  // что нет условия where

    select custTable                              // Не может использовать кэш, потому
        where custTable.AccountNum > '1101';      // что в where не предикат равенства (==)

    select custTable                              // Будет использован кэш, даже если в
        where custTable.AccountNum == '1101'      // where будет больше предикатов, чем
        && custTable.CustGroup == '20';           // полей в первичном ключе
                                                  // Предполагается, что записи были уже
                                                  // закэшированы ранее. См. также
                                                  // следующий пример
}
```

Примечание. Индекс по полю *RecId*, значения которого всегда уникальны в таблице, может быть указан в свойстве таблицы *PrimaryIndex*. Таким образом, вы можете задействовать кэширование, используя поле *RecId*.

Следующий пример показывает, как работают улучшенные механизмы кэширования в Microsoft Dynamics AX 2012, когда в выражении *where* запроса содержится больше полей, чем в уникальном индексе.

```
static void whenRecordDoesGetCached(Args _args)
{
    CustTable custTable,custTable2;

        // Используем демо-данные Contoso
        // Следующее выражение select не приведет к кэшированию результата выборки
```

```
        // при использовании кэша типа found, потому что выборка не вернет записи
        // Результат был бы закэширован при настройке кэширования FoundAndEmpty

        select custTable
            where custTable.AccountNum == '1101'
            && custTable.CustGroup == '20';

        // Следующий запрос приведет к кэшированию выбранной записи

        select custTable
            where custTable.AccountNum == '1101';

        // Этот запрос также заполнит кэш, потому что выборка вернет запись

        select custTable2
            where custTable2.AccountNum == '1101'
            && custTable2.CustGroup == '10';

            // Если вы перезапустите job, то все данные будут возвращены из кэша
}
```

Следующий пример кода **X++ показывает, как в среде времени выполнения** Microsoft Dynamics AX работает кэширование по уникальным индексам. В стандартном приложении первичным ключом в таблице *InventDim* является *InventDimId*, а комбинация полей (*inventBatchId*, *wmsLocationId*, *wmsPalletId*, *inventSerialId*, *inventLocationId*, *configId*, *inventSizeId*, *inventColorId* и *inventSiteId*) составляет на таблице уникальный индекс.

Примечание. Этот пример основан на Microsoft Dynamics AX 2012. В версии Microsoft Dynamics AX 2012 R2 индекс был изменен.

```
static void UtilizeUniqueIndexCache(Args _args)
{
    InventDim InventDim;
    InventDim inventdim2;

    select firstonly * from inventdim2;

    // Возьмет данные из кэша, потому что в качестве предиката используется только
    // первичный ключ

    select inventDim
    where inventDim.InventDimId == inventdim2.InventDimId;
```

```
    info(enum2str(inventDim.wasCached()));

    // Возьмет данные из кэша, потому что набор полей в выражении where
    // соответствует уникальному индексу для таблицы InventDim, а значения полей
    // указывают на ту же запись, которая уже была выбрана по первичному ключу

    select inventDim
    where inventDim.inventBatchId  == inventDim2.inventBatchId
      && inventDim.wmsLocationId   == inventDim2.wmsLocationId
      && inventDim.wmsPalletId     == inventDim2.wmsPalletId
      && inventDim.inventSerialId  == inventDim2.inventSerialId
      && inventDim.inventLocationId == inventDim2.inventLocationId
      && inventDim.ConfigId        == inventDim2.ConfigId
      && inventDim.inventSizeId    == inventDim2.inventSizeId
      && inventDim.inventColorId   == inventDim2.inventColorId
      && inventDim.inventSiteId    == inventDim2.inventSiteId;
    info(enum2str(inventDim.wasCached()));

    // Не сможет использовать кэш, потому что выражение where не соответствует
    // списку полей в уникальном индексе или в первичном ключе

    select firstonly inventDim
    where inventDim.inventLocationId   == inventDim2.inventLocationId
      && inventDim.ConfigId            == inventDim2.ConfigId
      && inventDim.inventSiteId        == inventDim2.inventSiteId;
    info(enum2str(inventDim.wasCached()));
}
```

Среда времени выполнения Microsoft Dynamics AX обеспечивает выборку всех полей записи, прежде чем та будет закэширована. Таким образом, если среда времени выполнения не может найти запись в кэше, то перед отправкой базе данных запроса *SELECT* она всегда изменяет список полей в запросе, чтобы он включал все поля таблицы. Следующий пример кода X++ иллюстрирует это поведение.

```
static void expandingFieldList(Args _args)
{
    CustTable custTable;

    select CreditRating // список полей будет расширен до всех полей таблицы
        from custTable
        where custTable.AccountNum == '1101';
}
```

За счет расширения списка полей гарантируется, что запись, выбранная из БД, будет содержать значения для всех полей, прежде чем она будет помещена в кэш. И хотя производительность при выборке всех полей уступает производительности при выборке лишь нескольких из них, такие расходы считаются оправданными, поскольку рост производительности, обусловленный использованием кэша, значительно перекрывает потери производительности от его заполнения.

Совет. Вы можете отключить использование кэша путем вызова табличного метода *disableCache* с параметром *true*. Это заставит среду времени выполнения производить поиск записи в базе данных, а также не даст среде времени выполнения расширять список выбираемых полей.

Среда времени выполнения Microsoft Dynamics AX создает и использует кэш записей и на уровне клиента, и на уровне сервера. Кэш на стороне клиента используется локально только одним приложением клиента, а кэш на стороне сервера совместно используется всеми подключениями к серверу приложений, включая подключения от Windows-клиентов Microsoft Dynamics AX, веб-клиентов, .NET Business Connector и любые другие подключения.

Используемый кэш зависит от того, с какого уровня был сделан запрос поиска записи. Если поиск выполняется на уровне сервера, то используется кэш на стороне сервера. Если поиск выполняется с уровня клиента, то сначала выполняется поиск в локальном кэше клиента; если требуемая запись не найдена, то поиск выполняется в кэше на стороне сервера. Если и в нем отсутствует требуемая запись, то выполняется запрос к базе данных. Когда из базы данных запись возвращается на уровень сервера, а от него – на уровень клиента, то данная запись сохраняется и в кэше на стороне сервера, и в кэше на стороне клиента.

В версии Microsoft Dynamics AX 2009 клиент хранил в кэше до 100 записей на таблицу, а AOS – до 2000 записей на таблицу, и эти параметры нельзя было изменить. В Microsoft Dynamics AX 2012 вы можете настраивать размер кэша с помощью формы Конфигурация сервера (Администрирование системы > Настройка > Конфигурация сервера). Более подробную информацию вы можете найти в разделе «Параметры настройки производительности» далее в этой главе.

Кэши реализуются с использованием AVL-деревьев (являющихся сбалансированными бинарными деревьями), на которые накладывается ограничение, предотвращающее неограниченный рост дерева. Когда при вставке новой записи в кэш достигается допустимый максимум, среда времени выполнения удаляет приблизительно от 5% до 7% самых старых записей путем сканирования всего дерева. Следовательно, в сценариях, когда выполняется повторный поиск тех же записей и предполагается, что они находятся в кэше, может наблюдаться снижение производительности, если кэш постоянно полон. Это снижение производительности будет происходить не только потому, что записи не будут найдены в кэше, поскольку они были удалены как самые старые, что приводит к выполнению их поиска в базе данных, но еще и вследствие постоянного сканирования дерева в целях удаления устаревших записей. В следующем примере кода все записи таблицы *SalesTable* считываются дважды, и для каждого считывания выполняется поиск связанной записи в таблице *CustTable*. Если бы данный код X++ выполнялся на сервере и число записей в таблице *CustTable* было бы больше 2000 (предположим, что кэш на сервере настроен на такое максимальное количество записей), то самые старые записи были бы удалены из кэша, и после первого цикла считывания кэш не содержал бы все записи из таблицы *CustTable*. Когда код вновь пробегает по записям таблицы *SalesTable*, в кэше могут находиться не все записи, и выбор записи таблицы *CustTable* может привести к поиску записи в базе данных. Следовательно, сценарий выполнился бы гораздо быстрее при менее чем 2000 записей в таблице базы данных.

```
static void AgingScheme(Args _args)
{
    SalesTable salesTable;
    CustTable custTable;

    while select salesTable order by CustAccount
    {
        select custTable            // заполняем кэш
            where custTable.AccountNum == salesTable.CustAccount;

            // здесь идет остальной код
    }

    while select salesTable order by CustAccount
    {
        select custTable            // записей может не быть в кэше
```

```
            where custTable.AccountNum == salesTable.CustAccount;

        // здесь идет остальной код
    }
}
```

Важно. Проводите тестирование улучшения производительности за счет кэширования записей только на базе данных, чей размер и распределение данных схожи с рабочей базой. (Аргументация приведена в примере выше.)

Перед тем как среда времени выполнения Microsoft Dynamics AX выполняет поиск, вставляет, обновляет или удаляет записи в кэше, она задействует взаимно исключающую блокировку, которая не снимается, пока операция не будет закончена. Это означает, что два процесса, исполняемые на одном и том же сервере не могут выполнять данные операции в одно и тоже время; в любой момент времени только один процесс может удерживать блокировку, а остальные процессы блокируются. Блокировка происходит только тогда, когда среда времени выполнения работает с кэшем на уровне сервера. Поэтому, несмотря на то что возможности кэширования, поддерживаемые средой времени выполнения, являются весьма полезными, неправильное их использование может оказаться хуже, чем полный отказ от кэширования. Если вы можете повторно использовать уже заполненный буфер записей, следует так и поступить. Следующий код X++ демонстрирует выборку одной и той же записи: второй оператор *select* использует кэш, хотя можно было использовать первый, уже заполненный, табличный буфер.

```
static void ReuseRecordBuffer(Args _args)
{
    CustTable       custTable;
    CurrencyCode    myCustCurrency;
    CustGroupId     myCustGroupId;
    PaymTermId      myCustPaymTermId;

    // Плохой шаблон кодирования

    myCustGroupId = custTable::find('1101').CustGroup;
    myCustPaymTermId = custTable::find('1101').PaymTermId;
    myCustCurrency = custTable::find('1101').Currency;
```

```
    // Для этих выборок будет использован кэш, но куда эффективнее было бы повторно
    // использовать выбранную запись, поскольку даже выборка из кэша не «бесплатна»
    // Хороший шаблон кодирования

    custTable           = CustTable::find('1101');
    myCustGroupId       = custTable.CustGroup;
    myCustPaymTermId    = custTable.PaymTermId;
    myCustCurrency      = custTable.Currency;
}
```

Кэширование запросов с объединением по уникальному индексу

Кэширование запросов с объединением по уникальному индексу появилось в Microsoft Dynamics AX 2012, оно позволяет кэшировать дочерние и родительские таблицы в случае наследования таблиц, объединения таблиц с отношением один-к-одному и выборкой по уникальному индексу или комбинацию обоих вариантов. Основным ограничением в случае этого типа кэширования является то, что вы можете выбрать из кэша лишь одну запись по уникальному индексу и можете объединять таблицы в запросе только по полям уникального индекса.

Следующий код иллюстрирует все три возможных ситуации.

```
public static void main(Args args)
{
    SalesTable      header;
    SalesLine       line;
    DirPartyTable   party;
    CustTable       customer;
    int             i;

        // кэширование записей родительской и дочерних таблиц

        for (i=0 ; i<1000; i++)
            select party where party.RecId == 5637144829;

        // кэширование объединений таблиц с отношением 1:1

        for (i=0 ; i<1000; i++)
            select line
            join header
            where line.RecId == 5637144586
                && line.SalesId == header.SalesId;

        // комбинация кэширования родительской и дочерних таблиц, а также
```

```
        // объединения 1:1

        for (i=0 ; i<1000; i++)
            select customer
            join party
            where customer.AccountNum == '4000'
                && customer.Party == party.RecId;
}
```

Кэширование вида *EntireTable*

В дополнение к трем ранее описанным методам кэширования – *Found, FoundAndEmpty* и *NotInTTS* – вы можете использовать для таблицы четвертый вариант – *EntireTable*. Данный вариант задействует кэш на основе набора записей. Его использование приводит к тому, что AOS при первой выборке записи из таблицы делает копию таблицы из базы данных путем выбора всех записей из нее и вставки их во временную таблицу. Поэтому время отклика первого процесса, который читает записи из этой таблицы, может быть более продолжительным, так как среде времени выполнения нужно время для выборки всех записей из таблицы. Последующие запросы *select* читают данные из кэша сервера приложения вместо обращения к базе данных.

Временная таблица обычно является локальной для того процесса, который ее использует, но кэш *EntireTable* совместно используется всеми процессами, которые обращаются к одному и тому же серверу AOS. Каждая компания (определяется в поле *DataAreaId*) имеет свой кэш уровня таблицы, поэтому два процесса, запрашивающие записи из одной и той же таблицы, но из различных компаний, используют различные кэши, и из-за необходимости заполнения кэша при первом чтении, у обоих из них время отклика может быть больше обычного.

Кэш *EntireTable* может находиться только на уровне сервера. Когда с уровня клиента выполняется запрос записей из таблицы, которая находится в кэше *EntireTable*, таблица ведет себя так же, как при использования кэширования типа *Found*. Если выполняется запрос записи на уровне клиента, который удовлетворяет условиям поиска в кэше, то клиент сначала выполняет поиск в локальном кэше типа *Found*. Если запись в локальном кэше не найдена, то клиент отсылает запрос серверу AOS, который выполняет поиск в кэше типа *EntireTable*. Когда среда времени выполнения возвращает найденную запись клиенту, то она вставляется в локальный кэш типа *Found*. На сервере кэш типа *EntireTable* дополни-

тельно использует кэш типа *Found*, когда выполняется поиск записей по уникальному ключу.

Кэш типа *EntireTable* не используется, когда выполняется оператор *select*, в котором таблица с типом кэширования *EntireTable* объединяется с таблицей с иным типом кэширования. В данной ситуации весь оператор *select* передается на сервер базы данных. Однако кэш используется в выборках, где оператор *select* выбирает записи только к одной таблицы с типом кэширования *EntireTable,* или когда в запросе объединяются несколько таких таблиц.

Среда времени выполнения Microsoft Dynamics AX очищает кэш типа *EntireTable*, когда выполняются операции вставки, обновления или удаления записей из этой таблицы. Поэтому производительность процесса, который следующим читает записи из данной таблицы, ухудшится в результате необходимости перечитать все записи таблицы в кэш. В дополнение к очистке собственного кэша, сервер AOS, который выполняет вставку, обновление или удаление записи, также информирует другие сервера приложения AOS о том, что они должны очистить кэш той же таблицы. Это не дает устаревшим и недостоверным данным быть кэшированными слишком долго в среде приложения Microsoft Dynamics AX в целом. В дополнение к данному механизму очистки кэша типа *EntireTable,* сервер приложения каждые 24 часа очищает все кэши типа *EntireTable*.

Поскольку при изменении записей в таблице с типом кэширования *EntireTable* происходит сброс кэша, избегайте использования данного типа кэширования для часто обновляемых таблиц. Повторное чтение всех записей в кэш приводит к потерям производительности, которые могут перевесить повышение производительности, достигнутое с помощью кэширования записей на уровне сервера. Вы можете переопределить настройку типа кэширования *EntireTable* для той или иной таблицы во время выполнения, когда настраиваете Microsoft Dynamics AX.

Даже если записи в конкретной таблице остаются неизменными достаточно долго, вам, возможно, удастся достичь лучшей производительности, не используя кэш типа *EntireTable*, если число записей в таблице слишком велико. Поскольку кэш типа *EntireTable* использует временные таблицы, то происходит переключение от хранения таблицы в памяти к ее хранению в файле, когда требования таблицы к памяти превышают предел в 128 килобайт. Это, конечно же, приводит к ухудшению производительности в процессе поиска записей. Кроме того, механизмы поиска в базах данных развиваются на протяжении многих лет и естественно работают бы-

стрее, чем механизм поиска, реализованный в среде исполнения Microsoft Dynamics AX. Может оказаться более правильным решением позволить базе данных выполнять поиск записей, чем настраивать и использовать кэш типа *EntireTable*, несмотря на то, что в данном случае будут использоваться дополнительные обращения к серверу базы данных. В Microsoft Dynamics AX 2012 вы можете настроить, сколько памяти будет отводиться на кэширование таблицы целиком, прежде чем кэш будет выгружен на диск. Это можно сделать на форме Администрирование > Настройка > Система > Конфигурация сервера.

Класс *RecordViewCache*

Объект *RecordViewCache* реализован как связанный список, позволяющий выполнять лишь последовательный поиск записей. Когда кэшируется большое число записей, производительность кэша падает из-за хранения данных в виде связанного списка, поэтому не следует использовать кэш этого типа для хранения более чем 100 записей. Оцените, насколько использование кэша эффективнее, чем выборка записей из базы данных, которая хотя и несет дополнительные накладные расходы, но задействует более оптимальные алгоритмы поиска. В частности, подумайте о том, сколько времени потребуется для поиска лишь подмножества записей; среде времени выполнения Microsoft Dynamics AX нужно будет постоянно сравнивать каждую запись в кэше с условиями выборки в выражении *where* оператора *select*, потому что для записей в этом типе кэша не доступны никакие индексы.

Вы можете использовать класс *RecordViewCache* из кода X++ для создания кэша на основе набора записей. Кэш инициируется при помощи следующего кода:

```
select nofetch custTrans where custTrans.accountNum == '1101';
recordViewCache = new RecordViewCache(custTrans);
```

В кэш помещаются записи, выбранные с помощью оператора *select*, который должен включать ключевое слово *nofetch*, чтобы запретить действительный выбор записей из базы данных. На самом деле записи выбираются, когда создается объект класса *RecordViewCache*, которому в качестве параметра передается этот табличный буфер. Теперь до уничтожения объекта класса *RecordViewCache* операторы *select* будут выбирать записи из созданного кэша, если они удовлетворяют условию *where*, определенному при его создании. В следующем примере кода показано, как создается и используется такой вид кэша.

```
public static void main(Args _args)
{
    InventTrans         inventTrans;
    RecordViewCache     recordViewCache;
    int countNone, countSold, countOrder;

    // определение того, какие записи нужно закэшировать

    select nofetch inventTrans
        where inventTrans.ItemId == '1001';

    // кэширование записей

    recordViewCache = new RecordViewCache(InventTrans);

    // использование кэша

    while select inventTrans
        index hint ItemIdx
        where inventTrans.ItemId == '1001' && inventTrans.StatusIssue == StatusIssue::OnOrder
    {
        countOrder++;

        // здесь идет дополнительный код
    }

    // Допустим, следующий фрагмент кода должен быть выполнен после первого
    // оператора while select и перед вторым оператором while select

    // здесь, собственно, и идет этот дополнительный фрагмент кода

    // Снова используем кэш

    while select inventTrans
        index hint ItemIdx
        where inventTrans.ItemId == '1001' && inventTrans.StatusIssue == StatusIssue::Sold
    {
        countSold++;
        // здесь идет дополнительный код
    }
    info('OnOrder Vs Sold = '+int2str(countOrder) + ' : ' + int2str(countSold));
}
```

Такой вид кэша может создаваться только на уровне сервера. Определяемый оператор *select* в условии *where* может содержать только предикаты равенства (==) и доступен только процессу, который создает объект кэша. Если буфер записей, используемый для создания объекта кэша, является временной таблицей или же он использует кэширование типа *EntireTable*, то объект класса *RecordViewCache* не создается.

Если записи таблицы, кэшированной в объекте *RecordViewCache*, также кэшируются на уровне отдельных записей, то среда времени выполнения может использовать оба кэша. Если оператор *select* выполняется для таблицы, кэшированной методом *Found*, и оператор *select* удовлетворяет требованиям поиска в кэше *Found*, то среда времени выполнения сначала производит поиск в данном кэше. Если ничего не обнаружено и оператор *select* также удовлетворяет требованиям поиска в объекте *RecordViewCache*, то среда времени выполнения использует запись из объекта *RecordViewCache* и обновляет кэш *Found* после извлечения записи.

Вставки, обновления и удаления записей, которые соответствуют критерию кэша, отражаются в кэше в то же самое время, когда операторы языка манипулирования данными (Data Manipulation Language, DML) отсылаются на сервер базы данных. Записи в кэш всегда вставляются в конец связанного списка. Риск, связанный с данным поведением, заключается в том, что может возникнуть ситуация бесконечного цикла, когда прикладная логика проходит по записям в кэше, и в то же самое время выполняется вставка новых записей, которые удовлетворяют критерию кэша.

Изменения, внесенные в записи в объекте *RecordViewCache*, не могут быть отменены. Если существует один или более объект *RecordViewCache* и выполняется операция *ttsabort* или генерируется исключение, которое приводит к откату транзакции базы данных, то объекты *RecordViewCache* будут все еще содержать информацию без учета отката. Поэтому время жизни любого созданного объекта *RecordViewCache*, записи которого будут изменяться в прикладной логике, не должно превышать область видимости транзакции, в которой он модифицируется. Объект класса *RecordViewCache* должен объявляться в методе, который начинает выполняться после начала транзакции. В случае отката объект и кэш будут автоматически удалены.

SysGlobalObjectCache и SysGlobalCache

Microsoft Dynamics AX 2012 предоставляет два механизма, которые вы можете использовать для кэширования глобальных переменных с целью

улучшения производительности: *SysGlobalObjectCache (SGOC)* и *SysGlobalCache*. *SGOC* является новой возможностью MicrosoftDynamics AX 2012 и играет важную роль в повышении производительности.

SGOC является глобальным кэшем, расположенным на сервере AOS, а не просто кэшем для одной отдельной сессии. Вы можете использовать этот кэш для сокращения циклов приема-передачи между сервером и базой данных или для хранения промежуточных результатов вычислений. Данные, сохраненные в одной пользовательской сессии, становятся доступны всем пользователям. Более подробную информацию о SGOC вы можете найти в статье «Using SysGlobalObjectCache (SGOC) and understanding its performanceimplications» в блоге команды Microsoft Dynamics AX, занимающейся вопросами производительности (*http://blogs.msdn.com/b/axperf/archive/2011/12/29/using-sysglobalobjectcache-sgoc-and-understanding-it-s-performance-implications.aspx*).

SysGlobalCache использует объект *map* для хранения информации, которая целиком привязана к текущей сессии. Однако, если вы используете этот способ кэширования, следует учесть определенные нюансы клиент-серверного взаимодействия. Если вы используете *SysGlobalCache* с помощью класса *ClassFactory*, то фактически используются два экземпляра глобального кэша: один на сервере, а другой – на клиенте, и обращение из кода идет к экземпляру, расположенному на том же уровне, где выполняется вызывающий код. Если вы используете *SysGlobalCache* напрямую, то он выполняется на том же уровне, где был создан экземпляр класса. Если вы используете *SysGlobalCache* с помощью класса *Info* или *Application*, то он располагается, соответственно, на клиенте либо на сервере, поэтому при доступе с другого уровня (например, при доступе с клиента с помощью класса *Application* к серверному *SysGlobalCache*) производительность снижается из-за дополнительных циклов приема-передачи между клиентом и сервером. Более подробную информацию вы можете найти в разделе «Using Global Variables» по адресу: *http://msdn.microsoft.com/en-us/library/aa891830.aspx*.

Ограничение списка полей выборки

Большинство операторов *select* в Microsoft Dynamics AX извлекают все поля записи несмотря на то, что в действительности используются значения только нескольких из них. Главная причина такого стиля кодирования заключается в том, что среда времени выполнения Microsoft Dynamics AX не сообщает об ошибках компиляции и времени выполнения, если вы-

полняется обращение к полю буфера записей, значение которого не было извлечено из базы данных запросом. Из-за нормализации модели данных Microsoft Dynamics AX 2012 и появления иерархий таблиц ограничение списка полей выборки в запросах стало еще более важным, чем это было в Microsoft Dynamics AX 2009, в частности, для полиморфных таблиц. В режиме *ad hoc* вы можете ограничить список полей в запросе; в этом режиме запрос ограничивается только таблицей (или таблицами), на которую(ые) в запросе есть явные ссылки. Прочие таблицы в иерархии исключаются из запроса. Это обеспечивает важное преимущество с точки зрения производительности за счет ограничения числа объединений между таблицами выше и ниже по иерархии.

Примечание. Базовая таблица в иерархии всегда выбирается независимо от того, к каким полям осуществляется доступ.

Следующий пример демонстрирует эффекты запросов:

```
static void AdHocModeSample(Args _args)
{
    DirPartyTable dirPartyTable;
    CustTable custTable;
    select dirPartyTable join custTable where dirPartyTable.RecId==custTable.Party;

    /* Приведет к следующему запросу к БД:
    SELECT T1.NAME,
    T1.LANGUAGEID,

--<...прочие поля удалены для лучшей читаемости. Выбираются все поля из всех таблиц>

    T9.MEMO FROM DIRPARTYTABLE T1 LEFT OUTER JOIN DIRPERSON T2 ON (T1.
RECID=T2.RECID) LEFT OUTER JOIN DIRORGANIZATIONBASE T3 ON (T1.RECID=T3.RECID)
LEFT OUTER JOIN DIRORGANIZATION T4 ON (T3.RECID=T4.RECID) LEFT OUTER JOIN
OMINTERNALORGANIZATION T5 ON (T3.RECID=T5.RECID) LEFT OUTER
JOIN OMTEAM T6 ON (T5.RECID=T6.RECID) LEFT OUTER JOIN OMOPERATINGUNIT T7
ON (T5.RECID=T7.RECID) LEFT OUTER JOIN COMPANYINFO T8 ON (T5.RECID=T8.RECID)
CROSS JOIN CUSTTABLE T9 WHERE ((T9.DATAAREAID='ceu') AND (T1.RECID=T9.PARTY))

    Ограничение списка полей приведет к тому, что AOS Microsoft Dynamics AX 2012 отправит запрос только по явно выбираемой таблице.
    Следующий запрос:*/
```

```
    select RecId from dirPartyTable exists join custTable where dirPartyTable.
RecId==custTable.Party;
    /*
Приведет к тому, что на SQL Server уйдет такой запрос:

    SELECT T1.RECID, T1.INSTANCERELATIONTYPE FROM DIRPARTYTABLE T1 WHERE
EXISTS (SELECT 'x' FROM CUSTTABLE T2 WHERE ((T2.DATAAREAID='ceu') AND (T1.
RECID=T2.PARTY)))
    */
}
```

Есть дополнительные способы ограничить список полей и число объединений таблиц в запросах через пользовательских интерфейс. Эти способы детально описаны в конце этой главы.

Следующий код X++, который выбирает только значение поля *AccountNum* таблицы *CustTable*, проверяет значение поля *CreditRating* и устанавливает значение поля *CreditMax*. Выполнение такого кода не приведет к сбою, поскольку среда времени выполнения не определяет, что поле не было выбрано.

```
static void UpdateCreditMax(Args _args)
{
    CustTable custTable;

    ttsBegin;
    while select forupdate AccountNum from custTable
    {
        if (custTable.CreditRating == '')
        {
            custTable.CreditMax = custTable.CreditMax + 1000;
            custTable.update();
        }
    }
    ttsCommit;
}
```

Данный код добавляет 1000 к значению поля *CreditMax* записей таблицы *CustTable*, у которых поле *CreditRating* пусто. Однако простое добавление *CreditRating* и *CreditMax* в список выбираемых полей не решит проблему: прикладная логика все равно может неправильно обновлять другие поля, потому что метод *update* таблицы может использовать и устанавливать другие поля обновляемой записи.

Важно. Вы могли бы просмотреть в коде метода *update*, какие еще поля он использует, и затем добавить их в список выбираемых полей, но вскоре возникнут новые проблемы. Например, если вы измените метод *update*, включив в него прикладную логику, которая использует дополнительные поля, то вы можете быть не в курсе того, что также нужно изменить код X++ из предыдущего примера.

Ограничение списка полей в выборке дает определенный выигрыш в производительности, потому что сокращает объем данных, пересылаемых от сервера базы данных серверу AOS. Выигрыш будет еще больше, если выбираемые поля могут быть получены из индексов без необходимости выбирать данные из самой таблицы или если за счет ограничения полей в выборке вы ограничиваете количество объединений в случае использования наследования таблиц. Такие оптимизации производительности и написание соответствующих запросов *select* безопасны, когда вы используете выбранные данные в рамках контролируемой области, такой как отдельный метод. Табличный буфер должен быть объявлен локально и не должен передаваться в другие методы в качестве параметра.

Любой разработчик, изменяющий код X++, может легко увидеть, что в коде используются лишь некоторые из полей выборки, и действовать соответственно. Однако, чтобы по-настоящему получить выигрыш от сокращения списка полей выборки, нужно быть в курсе того, что среда времени выполнения Microsoft Dynamics AX иногда автоматически добавляет дополнительные поля к списку выборки, прежде чем отправить запрос базе данных. Один из примеров объяснялся ранее в этой главе, в разделе «Кэширование». В том примере среда времени выполнения расширяла список полей выборки до всех полей записи, если оператор *select* удовлетворял условиям помещения выбранной записи в кэш.

В следующем коде X++ вы можете увидеть, как среда времени выполнения Microsoft Dynamics AX добавляет дополнительные поля. Код вычисляет суммарный баланс по всем клиентам из группы клиентов *20* и конвертирует полученную сумму в основную валюту компании. Метод *amountCur2MST* конвертирует значение в валюте, указанной в поле *CurrencyCode*, в валюту компании.

```
static void BalanceMST(Args _args)
{
    CustTable       custTable;
    CustTrans       custTrans;
    AmountMST       balanceAmountMST = 0;

    while select custTable
                where custTable.CustGroup == '20'
            join custTrans
                where custTrans.AccountNum == custTable.AccountNum
    {
        balanceAmountMST += Currency::amountCur2MST(custTrans.AmountCur,
                                                    custTrans.CurrencyCode);
    }
}
```

Когда оператор *select* передается базе данных, он выбирает все поля из таблиц *CustTable* и *CustTrans* несмотря на то, что использоваться будут лишь поля *AmountCur* и *CurrencyCode* из таблицы *CustTrans*. В результате из базы данных выбирается больше 100 полей *CustTrans*.

Список полей может быть оптимизирован за счет выбора только полей *AmountCur* и *CurrencyCode* из таблицы *CustTrans* и, например, только поля *AccountNum* из таблицы *CustTable*, как показано в следующем коде.

```
static void BalanceMST(Args _args)
{
    CustTable       custTable;
    CustTrans       custTrans;
    AmountMST       balanceAmountMST = 0;

    while select AccountNum from custTable
                where custTable.CustGroup == '20'
            join AmountCur, CurrencyCode from custTrans
                where custTrans.AccountNum == custTable.AccountNum
    {
        balanceAmountMST += Currency::amountCur2MST(custTrans.AmountCur,
                                                    custTrans.CurrencyCode);
    }
}
```

Как уже указывалось, среда времени выполнения расширяет список полей выборки с трех полей, показанных в предыдущем примере кода X++, до пяти, поскольку в список добавляются поля, используемые для обнов-

ления записей. Эти поля добавляются даже в том случае, если в операторе не содержится ни ключевое слово *forupdate*, ни какое-либо из ключевых слов для обозначения конкурентной модели. Запрос, отправляемый базе данных, начинается так, как показано в следующем примере, где для обеих таблиц добавлено поле RECID.

```
SELECT A.ACCOUNTNUM,A.RECID,B.AMOUNTCUR,B.CURRENCYCODE,B.RECID FROM
CUSTTABLE A,CUSTTRANS B
```

Чтобы предотвратить выборку каких-либо полей из таблицы *CustTable*, вы можете переписать оператор *select* с использованием *exists join*, как показано здесь:

```
static void BalanceMST(Args _args)
{
    CustTable       custTable;
    CustTrans       custTrans;
    AmountMST       balanceAmountMST = 0;

    while select AmountCur, CurrencyCode from custTrans
        exists join custTable
            where custTable.CustGroup == '20' &&
                  custTable.AccountNum == custTrans.AccountNum
    {
        balanceAmountMST += Currency::amountCur2MST(custTrans.AmountCur,
                                                    custTrans.CurrencyCode);
    }
}
```

Этот код выбирает из таблицы *CustTrans* три поля (*AmountCur*, *CurrencyCode* и *RecId*), а из таблицы *CustTable* – ни одного.

Впрочем, не во всех ситуациях возможно переписать оператор с использованием *exists join*. В таких случаях предотвратить выборку каких-либо полей из таблицы можно за счет указания поля *TableId* как единственного поля в списке выборки для таблицы. Исходный пример можно было бы изменить следующим образом, чтобы включить поле *TableId*:

```
static void BalanceMST(Args _args)
{
    CustTable       custTable;
    CustTrans       custTrans;
    AmountMST       balanceAmountMST = 0;

    while select TableId from custTable
```

```
                where custTable.CustGroup == '20'
            join AmountCur, CurrencyCode from custTrans
                where custTrans.AccountNum == custTable.AccountNum
    {
        balanceAmountMST += Currency::amountCur2MST(custTrans.AmountCur,
                                                    custTrans.CurrencyCode);
    }
}
```

При выполнении этого кода среда времени выполнения Microsoft Dynamics AX отправит базе данных оператор *select* с таким списком полей:

```
SELECT B.AMOUNTCUR,B.CURRENCYCODE,B.RECID FROM CUSTTABLE A,CUSTTRANS B
```

Таким образом, если вы перепишите оператор *select* с использованием *exists join* или включите лишь *TableId* в качестве выбираемого поля, то отправляемый базе данных оператор *select* будет выбирать лишь три поля вместо более чем ста. Как видите, можно существенно улучшить производительность вашего приложения просто за счет переписывания запросов таким образом, чтобы они выбирали лишь необходимые поля.

Совет. С помощью настройки проверок рекомендаций (Best Practices) вы может сделать так, чтобы Microsoft Dynamics AX анализировала операторы *select* в коде X++ и выдавала рекомендации, нужно ли воспользоваться явно заданным списком полей выборки, на основании того, какие поля используются в методе. Чтобы включить такую проверку, в среде разработки выберите пункт меню Инструменты > Параметры > Разработка > Рекомендации. Убедитесь, что в диалоговом окне настройки рекомендаций отмечен пункт Проверка производительности AOS и уровень детализации сообщений установлен в Ошибки или Предупреждения.

Чтобы использовать *ad hoc*-режим запросов на формах, в AOT перейдите к узлу *Data Sources* нужной формы, выберите необходимый источник данных и установите свойство *OnlyFetchActive* в значение *Yes*, как показано на рис. 13-6. Эта настройка ограничивает число полей выборки лишь теми, которые используются элементами управления формы, и, таким образом, улучшает время отклика формы. Кроме того, если источник данных связан с полиморфной таблицей, то в объединениях таблиц в запросе будут участвовать только те таблицы, которые необходимы для выборки этих полей, вместо всех таблиц в рамках иерархии.

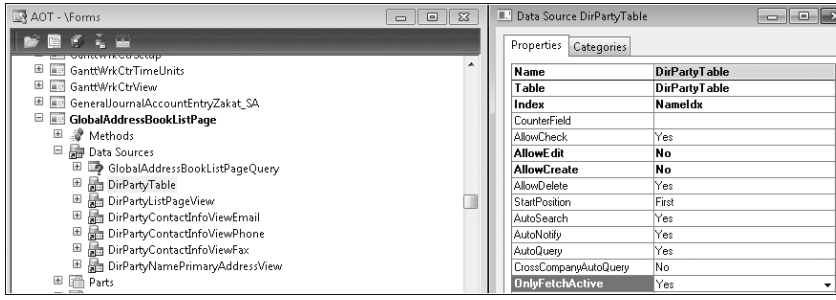

Рис. 13–6. *Использование OnlyFetchActive на странице списка*

Чтобы увидеть эффект от использования *ad hoc*-режима запросов, проведите следующий тест: создайте страницу списка, содержащую таблицу *DirPartyTable* в качестве источника данных и добавьте на страницу списка лишь три поля, например *Name*, *NameAlias* и *PartyNumber*. Установка свойства *OnlyFetchActive* в *No* приведет к выполнению следующего запроса, содержащего все поля из всех таблиц и объединения всех таблиц в иерархии.

```
SELECT T1.DEL_GENERATIONALSUFFIX,T1.NAME, T1.NAMEALIAS,T1.PARTYNUMBER,
/* Список полей сокращен для улучшения читаемости. Выбираются все поля из всех
таблиц. */
T8.RECID, FROM DIRPARTYTABLE T1 LEFT OUTER JOIN DIRPERSON T2 ON (T1.RECID=T2.
RECID) LEFT OUTER JOIN DIRORGANIZATIONBASE T3 ON (T1.RECID=T3.RECID) LEFT OUTER
JOIN DIRORGANIZATION T4 ON (T3.RECID=T4.RECID) LEFT OUTER JOIN OMINTERNALOR-
GANIZATION T5 ON (T3.RECID=T5.RECID) LEFT OUTER JOIN OMTEAM T6 ON (T5.RECID=T6.
RECID) LEFT OUTER JOIN OMOPERATINGUNIT T7 ON (T5.RECID=T7.RECID) LEFT OUTER
JOIN COMPANYINFO T8 ON (T5.RECID=T8.RECID)ORDER BY T1.PARTYNUMBER
```

Установка *OnlyFetchActive* в *Yes* приведет к выполнению более компактного и эффективного запроса.

```
SELECT T1.NAME,T1.NAMEALIAS, T1.PARTYNUMBER, T1.RECID, T1.RECVERSION,
T1.INSTANCERELATIONTYPE FROM DIRPARTYTABLE T1 ORDER BY T1.PARTYNUMBER
```

Чтобы улучшить производительность веб-элементов управления Корпоративного портала, использующих полиморфные таблицы в источниках данных, убедитесь, что вы также установили свойство *OnlyFetchActive* в *Yes* на источнике данных соответствующего набора данных.

Для использования *ad hoc*-режима в запросах, созданных в качестве элементов AOT, проделайте следующее.

1. Перейдите к нужному запросу, затем разверните узел *Data Sources* и в нем – узел соответствующего источника данных.
2. Щелкните по узлу *Fields* и затем установите свойство *Dynamic* в *No* (см. рис. 13-7).

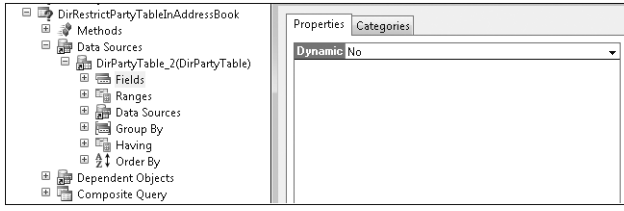

Рис. 13–7. Использование *ad hoc*-режима в запросах, созданных в AOT

3. Сократите список выбираемых полей до тех, которые действительно необходимы.

В стандартном приложении примером запроса с ограниченным списком полей выборки является *DirRestrictPartyTableInAddressBook*.

Выравнивание полей

Microsoft Dynamics AX для расширенных типов данных поддерживает левое и правое выравнивание значений. В Microsoft Dynamics AX 2012 практически все расширенные типы данных используют левое выравнивание, чтобы снизить эффект увеличения занимаемого данными объема, возникающий из-за двух- и трехбайтового кодирования символов Unicode. Кроме того, левое выравнивание также улучшает производительность за счет ускорения навигации по индексам. В случаях, когда критичным является порядок сортировки, можно использовать и правое выравнивание, но такие случаи должны быть лишь исключениями.

Параметры настройки производительности

В этом разделе приводится обзор наиболее важных настроек, улучшающих производительность установленной системы Microsoft Dynamics AX 2012.

Форма Администрирование SQL

Форма Администрирование SQL (рис. 13-8) предлагает доступ к набору возможностей SQL Server, которые не поддерживались в предыдущих версиях Microsoft Dynamics AX. Например, вы можете сжать таблицу или

выборочно изменить коэффициент заполнения. Форма Администрирование SQL находится в меню Администрирование системы > Периодические операции > База данных > Администрирование SQL.

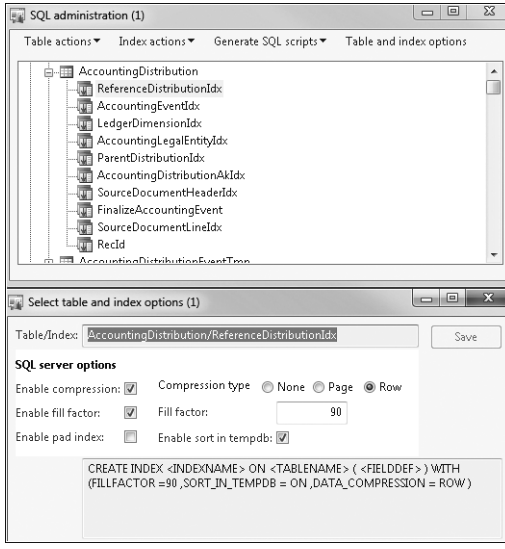

Рис. 13–8. Форма Администрирование SQL

Форма Конфигурация сервера

Несколько важных настроек производительности находятся на форме Конфигурация сервера. С помощью этой формы вы можете указать настройки оптимизации производительности, пакетных операций и кэширования. Форма Конфигурация сервера находится в меню Администрирование системы > Настройка > Система > Конфигурация сервера. Вот некоторые из наиболее важных настроек оптимизации производительности.

- **Максимальное количество таблиц в объединении.** Ограничивает число таблиц, которые могут участвовать в объединении. Слишком большое число объединений оказывает негативное влияние на производительность, особенно если оно идет по полям, которые не индексированы.
- **Коэффициент кэша записи клиента.** Определяет, сколько записей будет кэшироваться клиентом. Например, если серверная настройка кэширования таблицы из группы Main установлена в *2000*, а в этой

настройке вы укажете *20*, то клиент будет кэшировать 100 записей (2000/20).

- **Время ожидания для запросов, измененных пользователем.** Указывает таймаут в секундах, используемый для запросов, в которые с помощью формы *SysQueryForm* были добавлены дополнительные условия. Значение *0* указывает на отсутствие таймаута. Если время выполнения запроса превышает таймаут, то выводится соответствующее сообщение.

Вы также можете указать, является ли сервер AOS сервером пакетных заданий и сколько потоков сервер может использовать для их обработки. Хорошей формулой определения того, сколько потоков для обработки пакетных заданий может использовать сервер, является умножение на 2 числа процессорных ядер. Число потоков, которые может использовать сервер, зависит от выполняемых заданий. Для некоторых заданий сервер может использовать более двух потоков на ядро, однако такие случаи надо проверять индивидуально. Кроме того, вы можете указать, какое число записей может храниться в кэше, а также другие параметры кэша (см. рис. 13-9), такие как размер кэша типа *EntireTable* (в килобайтах), максимальное число объектов, хранящихся в *SGOC*, и число записей, которые могут быть закэшированы для каждой группы таблиц. При этом у каждого сервера могут быть свои настройки кэширования.

Настройка AOS

Утилита Microsoft Dynamics AX 2012 Server Configuration содержит настройки, которые вы можете использовать для улучшения производительности сервера AOS. Эта утилита доступна из меню Пуск > Администрирование > Microsoft Dynamics AX 2012 Server Configuration. Вот некоторые из наиболее важных настроек.

- **Вкладка Application Object Server.** Обычно настройки Enable Breakpoints To Debug X++ Code Running On This Server (Разрешить точки останова для отладки кода X++, выполняющегося на этом сервере) и Enable Global Breakpoints (Разрешить глобальные точки останова) должны быть выключены в рабочей системе как и Enable The Hot-Swapping Of Assemblies For Each Development Session (Включить горячую замену сборок для каждой сессии разработки). Все эти три настройки в случае включения могут существенно повлиять на производительность системы.

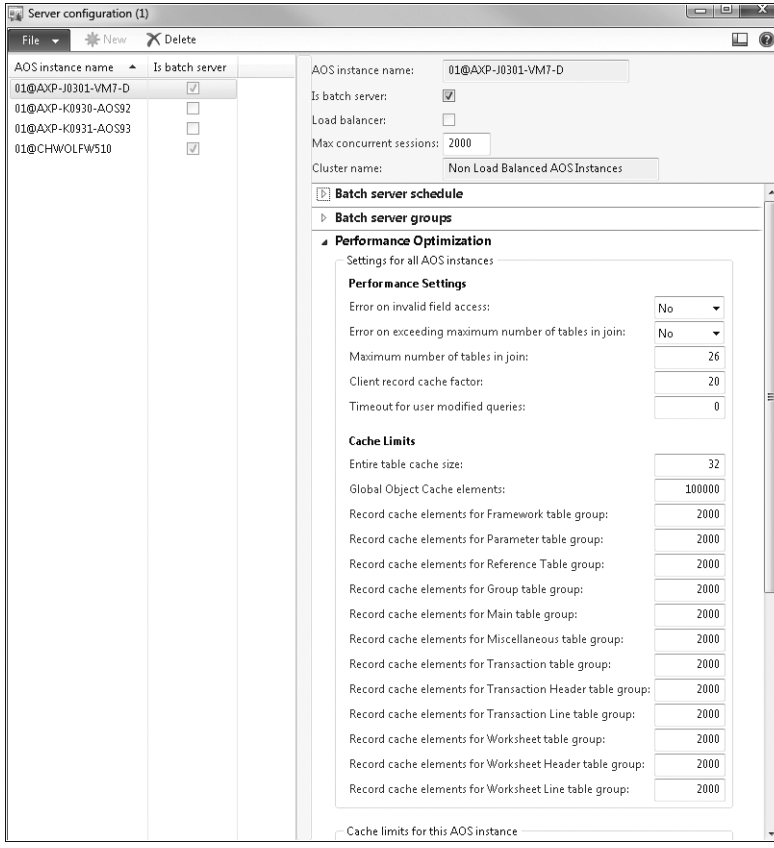

Рис. 13–9. Настройки кэширования на форме Конфигурация сервера

- **Вкладка Database Tuning.** В зависимости от используемых бизнес-процессов, увеличение значения Statement Cache (кэширование запросов) может как улучшить, так и ухудшить производительность. Эта настройка определяет, сколько запросов будет кэшировать AOS (кэшируются только сами запросы, а не результирующие выборки). Не следует изменять значение по умолчанию без проведения тщательного тестирования. Также по возможности не стоит изменять настройку Maximum Buffer Size (Максимальный размер буфера), потому что чем больше размер буфера, тем больше памяти выделяется на каждый буфер, что занимает несколько больше времени.

- **Вкладка Performance.** Если у вас используется несколько экземпляров AOS на одном сервере, то на этой вкладке можно указать маску используемых процессорных ядер, чтобы избежать соперничества экземпляров AOS за ресурсы процессора. Стоит учесть, что AOS в Microsoft Dynamics AX 2012 может эффективно использовать более восьми процессорных ядер.

Настройка клиента

Для настройки параметров клиента Microsoft Dynamics AX используется утилита Microsoft Dynamics AX Configuration, доступная в меню Пуск > Администрирование > Microsoft Dynamics AX 2012 Configuration. Если на вкладке Performance, в группе Cache Settings выбрать настройку Least Memory, то для экономии памяти загрузка некоторых DLL будет отложена до того момента, пока они не понадобятся. Эта настройка слегка снижает производительность, но она незаменима в случае использования терминального сервера, когда нужно увеличить число клиентских сессий, которые могут одновременно выполняться на терминальном сервере.

Производительность клиента

Вы можете использовать форму Параметры производительности клиента для централизованного отключения ряда возможностей, которые могут влиять на производительность. Эта форма доступа из меню Администрирование системы > Настройка > Система > Параметры производительности клиента. Более подробную информацию о настройках на этой форме вы можете найти в публикации «Microsoft Dynamics AX 2012: Client Performance Options» в блоге команды Microsoft Dynamics AX, занимающейся вопросами производительности (*http://blogs.msdn.com/b/axperf/archive/2011/11/07/ax2012-client-performance-options.aspx*).

Кэширование номерных серий

Общей рекомендацией является анализ всех используемых номерных серий для определения того, необходимо ли им быть непрерывными, и по возможности отказ от настройки Непрерывная в номерных сериях. Для всех номерных серий, которые не являются непрерывными, должно быть включено кэширование. В меню Управление организацией > Общие > Номерные серии щелкните дважды по нужной номерной серии и затем на экспресс-вкладке Производительность настройте предвари-

тельное выделение в зависимости от того, как часто эта номерная серия используется.

Подробное логирование

Подробное логирование изменений в базе данных и другие механизмы логирования, такие как транзакционный лог модуля продаж и маркетинга (Продажи и маркетинг > Настройка > Параметры модуля продаж и маркетинга) увеличивают нагрузку на базу данных и должны быть сокращены до абсолютно необходимого минимума.

Сводное планирование и закрытие склада

В Microsoft Dynamics AX 2012 производительность процессов сводного планирования и закрытия склада была оптимизирована. Оба процесса должны выполняться хотя бы с одним потоком-помощником, но предпочтительно использовать несколько потоков. Для сводного планирования на тестовых данных оптимальным оказалось использование восьми помощников. Другим способом улучшить производительность сводного планирования является выделение для него отдельного сервера AOS и изменение настроек сборщика мусора для работы только на клиенте. Для этого перейдите в каталог установки соответствующего AOS, откройте на редактирование файл Ax32Serv.exe.config, найдите следующий узел XML и измените значение атрибута на *false*:

```
<gcServer enabled="true" />
```

Шаблоны кодирования для повышения производительности

В этом разделе обсуждаются шаблоны кодирования, использование которых поможет вам в оптимизации производительности.

Выполнение кода X++ в виде CIL

Вы можете повысить производительность за счет выполнения кода X++ в виде common intermediate language (CIL). В общем случае, если сервис Microsoft Dynamics AX 2012 вызывается извне, то он выполняется в CIL. Пакетные задания, сервисы из группы сервисов AxClient и код, выполняемый с использованием метода *RunAs*, также выполняются в CIL. Для целей

выполнения кода в CIL существуют два метода в классе Global – *runClassMethodIL* и *runTableMethodIL*.

Выигрыш в производительности от исполнения кода X++ в CIL появляется, главным образом, за счет использования более эффективной сборки мусора в .NET. В зависимости от выполняемой логики производительность может улучшиться в диапазоне от 0% до 30%, в связи с чем необходимо проводить тестирование, чтобы определить, улучшится ли производительность вашего кода от исполнения его в CIL.

Эффективное использование параллельной обработки

В Microsoft Dynamics AX 2009 появился способ простой реализации параллельной обработки за счет использования инфраструктуры пакетных заданий. Соответствующие возможности были расширены в Microsoft Dynamics AX 2012. Для планирования работы пакетных заданий, параллельно выполняющих обработку, могут быть использованы три шаблона: создание наборов заданий, моделирование отдельных задач и выборка с вершины стека. У каждого шаблона есть свои достоинства и недостатки, которые обсуждаются в последующих разделах.

Более подробную информацию об инфраструктуре пакетных заданий вы можете найти в главе 18 . Примеры кода и дополнительную информацию о шаблонах создания пакетных заданий и их производительности вы можете найти в публикации «Batch Parallelism Microsoft Dynamics AX – Part I» в блоге команды Microsoft Dynamics AX, занимающейся вопросами производительности (*http://blogs.msdn.com/b/axperf/archive/2012/02/24/batch-parallelism-in-ax-part-i.aspx*). В следующих разделах приведены ссылки на дополнительные материалы.

Создание наборов заданий

При использовании наборов заданий вы создаете фиксированное число задач и разделяете работу между этими задачами за счет группировки рабочих заданий в наборы. Нагрузка, распределяемая на каждую задачу, должна быть одинаковой, насколько это возможно. Каждый рабочий поток обрабатывает набор рабочих элементов, а затем выбирает следующий набор. Этот шаблон хорошо работает, если все задачи в каждом наборе требуют примерно одного и того же времени на обработку. В идеальной ситуации каждый рабочий поток выполняет один и тот же объем работы.

Однако, в сценариях с переменной нагрузкой, вызванной различиями в структуре данных или используемом серверном оборудовании, этот

шаблон будет не самым эффективным. В таких ситуациях последние несколько потоков могут выполняться дольше, потому что они будут обрабатывать более обширные наборы заданий, чем другие потоки.

Пример кода, иллюстрирующий создание наборов заданий, вы можете найти в AOT, в методе Classes\FormletterServiceBatchTaskManager\createFormletterParmDataTasks().

Моделирование отдельных задач

При использовании моделирования отдельных задач обработка распараллеливается за счет создания отдельной задачи для каждого рабочего элемента, в результате чего между задачами и рабочими элементами получается соответствие один-к-одному. Это устраняет необходимость предварительного распределения задач. Из-за того что рабочие элементы обрабатываются рабочими потоками независимо, рабочая нагрузка распределяется более равномерно. Такой подход устраняет проблему, связанную с тем, что в набор объединяется большой набор рабочих задач, что в конечном итоге приводит к увеличению времени отклика пакетного задания.

Этот шаблон необязательно подойдет для обработки большого числа рабочих задач, потому что в результате у вас будет большое количество пакетных заданий. Накладные расходы инфраструктуры пакетных заданий, связанные с обработкой большого числа пакетов, довольно высоки, потому что эта инфраструктура должна проверять несколько условий, зависимостей и ограничений каждый раз, когда завершается один набор задач и нужно выбрать новый набор задач, готовых к выполнению.

Пример кода, иллюстрирующий этот шаблон, вы можете найти в блоге команды Microsoft Dynamics AX, занимающейся вопросами производительности (*http://blogs.msdn.com/b/axperf/archive/2012/02/25/batch-parallelism-in-ax-part-ii.aspx*).

Выборка с вершины стека

Одна из проблем, связанных с объединением рабочих задач в наборы, – неравномерное распределение нагрузки. С этой проблемой можно бороться за счет моделирования отдельных задач, однако это может привести к высокой дополнительной нагрузке на инфраструктуру пакетных заданий. Выборка с вершины стека – еще одна техника, которую можно использовать для решения проблемы неравномерного распределения рабочей нагрузки между пакетными заданиями. Впрочем, при большом числе рабочих задач она приводит к тем же проблемам, что и моделирование отдельных задач.

При использовании выборки с вершины стека создается фиксированное число задач, как при создании наборов, и устраняется необходимость предварительного распределения задач, как при моделировании отдельных задач. За счет отсутствия предварительного распределения задач этот шаблон не зависит от инфраструктуры пакетных заданий в части разделения рабочих задач, но вам понадобится поддерживать таблицу со статусами для отслеживания прогресса обработки рабочих задач. Поддержка такой таблицы сама по себе создает дополнительные накладные расходы, но они намного ниже, чем накладные расходы инфраструктуры пакетных заданий. После заполнения таблицы со статусами рабочие потоки начинают обработку, выбирая следующий доступный элемент из таблицы, и продолжают выполнение, пока не останется ни одного доступного рабочего элемента. При этом не возникает ситуации простоя одного рабочего потока при чрезмерной загруженности других. Для реализации выборки с вершины стека вам понадобится использовать хинты *PESSIMISTICLOCK* и *READPAST*. Совместное использовании этих хинтов позволяет рабочему потоку выбрать следующий рабочий элемент без риска оказаться заблокированным.

Пример кода, иллюстрирующий этот шаблон, вы можете найти в блоге команды Microsoft Dynamics AX, занимающейся вопросами производительности (*http://blogs.msdn.com/b/axperf/archive/2012/02/28/batch-parallelism-in-ax-part-iii.aspx*).

Инфраструктура SysOperation

В Microsoft Dynamics AX 2012 сделаны доступными концепции программирования и реализованы шаги по замене инфраструктуры RunBase. Используя приходящую на замену ей инфраструктуру SysOperation, вы можете запускать сервисы Microsoft Dynamics AX в различных режимах выполнения; кроме того, у инфраструктуры SysOperation есть и преимущества в плане производительности. В ней реализовано четкое разделение ответственности между уровнями, и выполнение происходит целиком на уровне сервера. Эти улучшения гарантируют минимизацию циклов приема-передачи между клиентом и сервером.

Примечание. В главе 14 вы можете найти дополнительную информацию об инфраструктуре SysOperation и примеры кода, в которых инфраструктура SysOperation сравнивается с инфраструктурой RunBase. Если вы не знакомы с инфраструктурой Sys-

Operation, то рекомендуется перед продолжением чтения этого раздела прочесть главу 14.

Инфраструктура SysOperation поддерживает четыре режима выполнения.

- **Синхронный.** Вы можете выполнять сервис на сервере в синхронном режиме. Клиент ожидает завершения процесса на сервере и лишь затем продолжает свою работу.
- **Асинхронный.** Вы выполняете необходимые настройки в контракте данных и затем выполняете код на сервере. Клиент, однако, продолжает отвечать на запросы пользователя и может продолжать работать. Этот режим также позволяет сократить число циклов приема-передачи между клиентом и сервером.
- **Надежный асинхронный.** Выполнение операций в этом режиме равнозначно их выполнению на пакетном сервере с той лишь разницей, что пакетные задания удаляются после завершения безотносительно того, успешно они завершились или нет; история пакетных заданий при этом сохраняется. Этот шаблон помогает формировать операции, которые задействуют среду времени выполнения пакетного сервера, но не полагаются на имеющиеся возможности его администрирования.
- **Запланированный пакет.** Этот режим используется для регулярно выполняемых пакетных заданий.

В следующем примере показывается, как рассчитать множество простых чисел. Пользователь вводит начальное и конечное значения (например, 1000000 и 1500000), затем сервис рассчитывает все простые числа в этом диапазоне значений. Этот пример продемонстрирует разницу во времени работы при использовании каждого из режимов выполнения. Пример состоит из двух классов (сервис и контакт данных), таблицы для сохранения результатов, job'а и перечисления для демонстрации запуска в различных режимах выполнения.

Примечание. Вместо job'а для выполнения операции обычно используются пункты меню. При использовании пункта меню инфраструктура SysOperation формирует необходимое диалоговое окно для заполнения контракта данных.

Следующий код содержит точку входа сервиса.

```
[SysEntryPointAttribute(true)]
public void runOperation(PrimeNumberRange data)
{
    PrimeNumbers primeNumbers;

    // Потоки обычно оказывают воздействие при выполнении в инфраструктуре
    // пакетных заданий с использованием надежного асинхронного режима либо режима
    // запланированного пакета

    int i, start, end, blockSize, threads = 8;
    PrimeNumberRange subRange;
    start = data.parmStart();
    end = data.parmEnd();
    blockSize = (end - start) / threads;
    delete_from primeNumbers;
    for (i = 0; i < threads; i++)
    {
        subRange = new PrimeNumberRange();
        subRange.parmStart(start);
        subRange.parmEnd(min(start + blockSize, end));
        subRange.parmLast(i == threads - 1);
        this.findPrimes(subRange);
        start += blockSize + 1;
    }
}
```

Следующий пример – это метод, выполняющийся по-разному в зависимости от выбранного вами режима выполнения.

Примечание. Если метод выполняется в режиме надежного асинхронного выполнения или запланированного пакета, то он также демонстрирует шаблон объединения заданий в набор, обсуждавшегося ранее в разделе «Создание наборов заданий».

```
[SysEntryPointAttribute(false)]
public void findPrimes(PrimeNumberRange range)
{
    BatchHeader batchHeader;
    SysOperationServiceController controller;
    PrimeNumberRange dataContract;
    if (this.isExecutingInBatch())
    {
```

```
        ttsBegin;
        controller = new SysOperationServiceController('PrimeNumberService',
'findPrimesWorker');
        dataContract = controller.getDataContractObject('range‘);

        dataContract.parmStart(range.parmStart());
        dataContract.parmEnd(range.parmEnd());
        dataContract.parmLast(range.parmLast());

        batchHeader = this.getCurrentBatchHeader();
        batchHeader.addRuntimeTask(controller, this.getCurrentBatchTask().RecId);
        batchHeader.save();
        ttsCommit;
    }
    else
    {
        this.findPrimesWorker(range);
    }
}
```

И, наконец, метод, который выполняет основную работу:

```
private void findPrimesWorker(PrimeNumberRange range)
{
    PrimeNumbers primeNumbers;
    int i;
    int64 time;

    for (i = range.parmStart(); i <= range.parmEnd(); i++)
    {
        if (this.isPrime(i))
        {
            primeNumbers.clear();
            primeNumbers.PrimeNumber = i;
            primeNumbers.insert();
        }
    }

    if (range.parmLast())
    {
        primeNumbers.clear();
        primeNumbers.PrimeNumber = -1;
        primeNumbers.insert();
    }
}
```

Ниже приведен код job'а, запускающего пример вычисления простых чисел во всех четырех режимах выполнения.

```
static void generatePrimeNumbers(Args _args)
{
    SysOperationServiceController controller;
    int i, ticks, ticks2, countOfPrimes;
    PrimeNumberRange dataContract;
    SysOperationExecutionMode executionMode;
    PrimeNumbers output;

<... код диалога для демонстрации режима выполнения ...>

    executionMode = getExecutionMode();
    controller = new SysOperationServiceController('PrimeNumberService', 'runOperation',
executionMode);
    dataContract = controller.getDataContractObject('data');

    dataContract.parmStart(1000000);
    dataContract.parmEnd(1500000);
    delete_from output;
    ticks = System.Environment::get_TickCount();
    controller.parmShowDialog(false);
    controller.startOperation();

    <... код показа времени выполнения для продемонстрированных режимов ...>
}
```

Выполнение этого кода четыре раза во всех четырех режимах выполнения дало следующие результаты.

- **Синхронный.** 35658 простых чисел найдено за 44,74 секунды. Однако пользователь не мог продолжать работать с системой во время расчетов.
- **Асинхронный.** 35658 простых чисел найдено за 46,93 секунд, однако, во время расчетов клиент не отвечал, и пользователь не мог продолжать свою работу.
- **Надежный асинхронный.** 35658 простых чисел найдено за 16,16 секунд с использованием параллельной обработки и мгновенным запуском пакетных заданий (как обсуждалось ранее в этой главе).

  Этот режим выполнения лишь запускает задания на пакетном сервере, но он не полностью схож с выполнением пакетных заданий; на форме

Пакетных заданий они появляются лишь временно. Другое ключевое отличие надежного асинхронного режима выполнения заключается в том, что хотя он и использует движок инфраструктуры пакетных заданий, но он не ограничен настройкой числа доступных потоков, указываемой на форме Конфигурации сервера. Пока у сервера есть доступные ресурсы, он будет продолжать параллельное выполнение надежных асинхронных заданий и также запускать выполнение новых заданий. Если вы запустите слишком много заданий, то можете излишне нагрузить сервер, с другой стороны, этот режим позволяет эффективно использовать многоядерные системы.

- **Запланированный пакет.** 35658 простых чисел найдено за 31,78 секунд. (Пакетные задания стартовали не мгновенно, из-за чего возникли различие между пакетным режимом и режимом надежного асинхронного выполнения.)

Вам также следует убедиться, что вы оставляете достаточно ресурсов процессора для обработки обычных пользовательских сессий. Обычно хорошей идеей является разделение нагрузки, создаваемой пакетными заданиями, и нагрузки, создаваемой обычными пользовательскими сессиями. Инфраструктура SysOperation предлагает дополнительный способ распараллеливания нагрузки с использованием бизнес-логики. К примеру, вы могли бы создать класс-обертку, который выполняет множество асинхронных вызовов бизнес-логики. Предположим, ваш класс-обертка разносит накладные по всем заказам определенного клиента. Вы могли бы создать диалоговое окно для выбора одного или нескольких кодов клиентов, для которых нужно разнести накладные. Сама по себе эта логика могла бы затем выполнять один вызов сервиса для каждого кода клиента, однако эти вызовы могут создать чрезмерную нагрузку на ваш сервер, если не использовать ее с осторожностью. Следующий код – это измененная версия предыдущего примера, показывающая, как мог бы выглядеть этот код.

Примечание. Обычно для определения параметров запуска используется диалоговое окно.

```
// В реальной ситуации эта обертка должна быть классам и вызываться из пункта меню,
// соответствующего необходимому режиму выполнения.
static void generatePrimeNumbersAsyncCallPattern(Args _args)
{
    SysOperationServiceController controller;
```

```
    int i, primestart, primeend,blockSize, threads = 8,countOfPrimes,ticks,ticks2;
    PrimeNumberRange subRange;
    PrimeNumberRange dataContract;
    PrimeNumbers output;

    primestart = 1000000;
    primeend = 1500000;

    blockSize = (primeend - primestart) / threads;

    delete_from output;

    ticks = System.Environment::get_TickCount();

    for (i = 0; i < threads; i++)
    {
        controller = new SysOperationServiceController('PrimeNumberServiceAsyncCallPattern',
        'runOperation', SysOperationExecutionMode::ReliableAsynchronous);
        dataContract = controller.getDataContractObject('data');

        dataContract.parmStart(primestart);
        dataContract.parmEnd(min(primestart + blockSize, primeend));
        dataContract.parmLast(i == threads - 1);

        controller.parmShowDialog(false);
        controller.startOperation();

        primestart += blockSize + 1;
    }

    <... код показа времени выполнения для продемонстрированных режимов ...>
}
```

Шаблоны проверки существования записи

В зависимости от того, какой шаблон вы используете, проверка существования записи может вызвать избыточные обращения к базе данных. Ниже приводится пример некорректного определения существования записи. Для каждой записи, выбираемой во внешнем цикле, базе данных отправляется другой оператор *select*, чтобы проверить наличие определенной записи в таблице *WMSJournalTrans*. Если в таблице *WMSJournalTable* находится 10000 записей, то данный пример приведет к отправке 10001 запроса базе данных.

```
static void existingJournal()
{
    WMSJournalTable    wmsJournalTable = WMSJournalTable::find('014119_117');
    WMSJournalTable    wmsJournalTableExisting;
    WMSJournalTrans    wmsJournalTransExisting;

    boolean recordExists()
    {
        boolean foundRecord;
        foundRecord = false;

        while select JournalId from wmsJournalTableExisting
            where wmsJournalTableExisting.Posted == NoYes::No
        {
            select firstonly wmsJournalTransExisting
                where wmsJournalTransExisting.JournalId ==
wmsJournalTableExisting.JournalId   &&
                      wmsJournalTransExisting.InventTransType ==
wmsJournalTable.InventTransType    &&
                      wmsJournalTransExisting.InventTransRefId ==
wmsJournalTable.InventTransRefId;
            if (wmsJournalTransExisting)
                foundRecord = true;
        }
        return foundRecord;
    }

    if (recordExists())
        info('Record Exists');
    else
        info('Record does not exist');
}
```

В следующем примере демонстрируется более удачный подход, работающий с куда меньшими накладными расходами. Этот шаблон приводит к отправке лишь одного запроса и одного цикла приема-передачи между сервером и базой данных.

```
static void existingJournal()
{
    WMSJournalTable    wmsJournalTable = WMSJournalTable::find('014119_117');
    WMSJournalTable    wmsJournalTableExisting;
    WMSJournalTrans    wmsJournalTransExisting;
```

```
    boolean recordExists()
    {
        boolean foundRecord;
        foundRecord = false;

        select firstonly wmsJournalTransExisting
        join wmsJournalTableExisting
        where wmsJournalTransExisting.JournalId ==
        wmsJournalTableExisting.JournalId &&
        wmsJournalTransExisting.InventTransType ==
        wmsJournalTable.InventTransType &&
        wmsJournalTransExisting.InventTransRefId ==
        wmsJournalTable.InventTransRefId &&
        wmsJournalTableExisting.Posted == NoYes::No;

        if (wmsJournalTransExisting)
            foundRecord = true;
        return foundRecord;
    }

    if (recordExists())
        info('Record Exists');
    else
        info('Record does not exist');
}
```

Выполнение запроса к базе данных не чаще, чем требуется

Зачастую один и тот же запрос выполняется по несколько раз. Даже если за счет кэширования удается несколько сократить общие накладные расходы, повторное выполнение одного и того же запроса все равно может очень серьезно сказаться на производительности, но есть способы легко избежать этих проблем. Как правило, проблемы связаны с выполнением методов *find*, которые выполняются повторно либо внутри циклов, либо внутри методов *exists*. Следующий пример кода демонстрирует цикл, в котором повторно выполняются вызовы метода *CustParameters::find*.

```
static void doOnlyNecessaryCalls(Args _args)
{
    LedgerJournalTrans      ledgerJournalTrans;
    LedgerJournalTable      ledgerJournalTable = LedgerJournalTable::find('000242_010');
    Voucher                 voucherNum = '';
```

```
    while select ledgerJournalTrans
        order by JournalNum, Voucher, AccountType
        where ledgerJournalTrans.JournalNum == ledgerJournalTable.JournalNum
            && (voucherNum == '' || ledgerJournalTrans.Voucher == voucherNum)
    {
        // Если цикл возвращает несколько записей, то ниже - потенциально ненужные
        // обращение к кэшу и вызов метода

        ledgerJournalTrans.PostingProfile = CustParameters::find().PostingProfile;

        // Далее идет дополнительная логика...
    }
}
```

Повторяющиеся вызовы *CustParameters::find* всегда возвращают один и тот же результат. Даже если результата выборки закэширован, эти вызовы создают дополнительную нагрузку. Для оптимизации производительности вы могли бы переместить вызов за пределы цикла, избавившись от повторных вызовов.

```
static void doOnlyNecessaryCallsOptimized(Args _args)
{
    LedgerJournalTrans      ledgerJournalTrans;
    LedgerJournalTable      ledgerJournalTable = LedgerJournalTable::find('000242_010');
    Voucher                 voucherNum = '';
    CustPostingProfile postingProfile = CustParameters::find().PostingProfile;

    while select ledgerJournalTrans
        order by JournalNum, Voucher, AccountType
        where ledgerJournalTrans.JournalNum == ledgerJournalTable.JournalNum
            && (voucherNum == '' || ledgerJournalTrans.Voucher == voucherNum)
    {
        // Если цикл возвращает несколько записей, то ненужные обращения
        // к кэшу и вызовы метода отсутствуют

        ledgerJournalTrans.PostingProfile = postingProfile;

        // Далее идет дополнительная логика...
    }
}
```

Когда два запроса лучше объединения таблиц в одном запросе

Для определенного рода запросов очень сложно или почти невозможно создать эффективный индекс. В основном это происходит при использовании оператора *OR* (||) применительно к нескольким полям. Следующий пример на SQL Server, как правило, приведет к объединению по индексу, которое потенциально менее эффективно, чем непосредственный поиск записи.

```
static void TwoQueriesSometimesBetterThenOne(Args _args)
{
    InventTransOriginId          inventTransOriginId = 5637201031;
    InventTransOriginTransfer    inventTransOriginTransfer;

    // Заметьте: в любой момент времени истинным может быть лишь одно из условий

    select firstonly inventTransOriginTransfer
        where inventTransOriginTransfer.IssueInventTransOrigin    == inventTransOriginId
            || inventTransOriginTransfer.ReceiptInventTransOrigin  == inventTransOriginId;
    info(int642str(inventTransOriginTransfer.RecId));
}
```

Использование двух запросов может привести к дополнительному циклу приема-передачи, но в идеальном случае следующий код приведет лишь к одному запросу. Кроме того, первый и второй запросы являются эффективными выборками по имеющемуся индексу.

```
static void TwoQueriesSometimesBetterThenOneOpt(Args _args)
{
    InventTransOriginId inventTransOriginId = 5637201031;
    InventTransOriginTransfer inventTransOriginTransfer;

    select firstonly inventTransOriginTransfer
        where inventTransOriginTransfer.IssueInventTransOrigin == inventTransOriginId;

    info(int642str(inventTransOriginTransfer.RecId));

    if(!inventTransOriginTransfer.RecId)
    {
        select firstonly inventTransOriginTransfer
            where inventTransOriginTransfer.ReceiptInventTransOrigin == inventTransOriginId;
        info(int642str(inventTransOriginTransfer.RecId));
    }
}
```

Трюки с индексированием

Новой возможностью для создания оптимизированных индексов является свойство *Included columns* (включенные столбцы). С включенными столбцами становится проще, к примеру, создавать покрывающие индексы для запросов, в которых список полей выборки ограничен или же используется агрегирование данных. Более подробную информацию о покрывающих индексах и индексах с включенными столбцами вы можете найти в разделе MSDN «Создание индексов с включенными столбцами» по адресу: *http://msdn.microsoft.com/ru-ru/library/ms190806.aspx*.

Для создания индекса с включенными столбцами установите свойство индекса *IncludedColumn* в *Yes*, как показано на рис. 13-10.

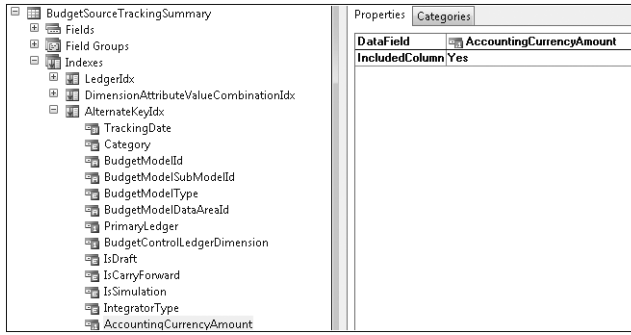

Рис. 13–10. Свойство индекса *IncludedColumn*

Другой, менее известной, возможностью является следующая: если вы добавляете в список ключевых полей индекса поле *dataAreaId*, то AOS уже не будет автоматически добавлять его в качестве первого поля в индексе, за счет чего можно оптимизировать выполнение определенных запросов. К примеру, запросы, которые не включают поле *dataAreaId* и используют прямое обращение к СУБД, приведут к сканированию индекса в случае, если используется этот индекс и поле *dataAreaId* является в нем первым. В общем случае пользоваться этой возможностью следует лишь в случае, если вы замечаете, что поля *dataAreaId* нет в запросе, и что SQL Server из-за этого выполняет сканирование индекса. Однако, без необходимости пользоваться этой возможностью не стоит, а если такая необходимость возникла, то всегда надо создавать для своих нужд новый индекс.

Когда использовать *firstfast*

Хинт *firstfast* добавляет *OPTION(FAST n)* в SQL-запрос и приводит к тому, что оптимизатор запросов отдает предпочтение индексу, хорошо подходящему под условия сортировки, потому что так запрос вернет первые записи как можно быстрее.

```
select firstfast salestable // Приводит к запросу SELECT <СписокПолей> FROM SALESTABLE
OPTION(FAST 1)
```

Примечание. Если вы сортируете по полям более чем из одной таблицы, то *OPTION(FAST n)* может не дать того выигрыша в производительности, на который вы рассчитываете.

Это ключевое слово автоматически используется для элементов управления *grid* на формах и может быть включено на источниках данных запросов в AOT. И насколько это ключевое слово может быть полезным – к примеру, на страницах списков, поддерживаемых запросами AOT, — настолько же оно может негативно повлиять на производительность запросов, потому что оно заставляет SQL Server оптимизировать планы исполнения запросов под условия сортировки вместо сокращения времени выполнения. Если в медленно выполняющемся запросе вы видите хинт *firstfast*, то попробуйте отключить его и проверьте, как изменится время отклика. Форма экспортного аккредитива/импортного инкассо является примером того, когда использование этой возможности приносит положительный результат. В AOT перейдите к Forms\BankLCExportListPage\Data Sources\BankLCExportListPage\Data Sources\SalesTable (SalesTable). На странице списка свойство *FirstFast* установлено в *No*, однако производительность улучшится, если выставить его в *Yes*.

Оптимизация страниц списков

Для оптимизации страниц списков вы можете экспериментировать с целым набором подходов. Зачастую запросы со страницы списков достаточно сложные и включают несколько источников данных. Сортировка выборки из объединенных таблиц может привести к падению производительности, поэтому для оптимизации постарайтесь избегать сортировки. Например, сокращение сортировки могло бы улучшить производительность формы Контакты. Запрос *smmContacts_NoFilter* (Forms/smmContactsListPage/DataSources/smmContacts_NoFilter) содержит две таблицы в сво-

ем выражении *Order by*. Для оптимизации производительности вы могли бы сортировать только по полю *ContactPerson.ContactForParty*.

Также для оптимизации производительности страницы списка вы можете использовать свойства *FirstFast* и *OnlyFetchActive*. Оба эти свойства были детально описаны ранее в этой главе.

Агрегирование полей для сокращения итераций перебора

Вместо того чтобы перебирать записи и агрегировать значения с помощью логики X++, вы зачастую можете агрегировать данные в запросе и тем самым сократить число итераций перебора и циклов приема-передачи между сервером и базой данных. Число итераций перебора, которые вы можете сократить, зависит, главным образом, от полей, по которым происходит агрегирование, и того, сколько записей может агрегироваться. Бывают ситуации, когда вы можете захотеть суммировать значения в вашем коде лишь при выполнении определенных условий.

В следующем примере сравниваются операции на основе наборов и операции на основе отдельных записей.

```
// В реальном коде следует использовать статический серверный метод, чтобы
// осуществлять доступ к данным на сервере.

public static void main(Args _args)
{
    TransferToSetBased      ttsb;
    RecordInsertList        ril = new RecordInsertList(tableName2id("TransferToSetBased"));
    Counter                 i;
    Counter                 tc;
    int                     myAggregate = 0;
    int                     my2ndAggregate;

    // сброс данных

    delete_from ttsb;

    // построчное заполнение

    tc = WinAPI::getTickCount();
    for(i=0;i<=1000;i++)
    {
        ttsb.clear();
```

```
        ttsb.Iterate=i;
        ttsb.Change=1;
        ttsb.Aggregate=5;
        ttsb.insert();
    }

    // заполнена 1000 записей, 1000 циклов приема-передачи

    for(i=1001;i<=2000;i++)
    {
        ttsb.clear();
        ttsb.Iterate=i;
        ttsb.Change=1;
        ttsb.Aggregate=5;
        ril.add(ttsb);
    }
    ril.insertDatabase();

    // заполнена 1000 записей, но циклов приема-передачи намного меньше
    // в зависимости от размера буфера, за один цикл вставлялось 20-150 записей

    ttsBegin;
    while select forupdate ttsb where ttsb.Iterate > 1000
{
    if(ttsb.Iterate >= 1100 && ttsb.Iterate <= 1300)
    {
        ttsb.Change = 10;
        ttsb.update();
        myAggregate += ttsb.Aggregate;
    }
    else if(ttsb.Iterate >= 1301 && ttsb.Iterate <= 1500)
    {
        ttsb.Change = 20;
        ttsb.update();
        my2ndAggregate += ttsb.Change;
    }
    else if(ttsb.Iterate >= 1501 && ttsb.Iterate <= 1700)
    {
        ttsb.Change = 30;
        ttsb.update();
        myAggregate += ttsb.Aggregate;
    }

    if(ttsb.Iterate > 1900)
```

```
        break;
}
ttsCommit;

// Цикл while сделал 1-900 выборок, было выполнено 600 отдельных операторов update
// переделка логики выше с использованием операций на основе наборов
// и агрегирования приводит к тому, что базе данных отправляется всего 6 запросов

update_recordSet ttsb setting change = 10 where ttsb.Iterate >= 1100 && ttsb.iterate <=
        1300;
update_recordSet ttsb setting change = 20 where ttsb.Iterate >= 1301 && ttsb.Iterate <=
        1500;
update_recordSet ttsb setting change = 30 where ttsb.Iterate >= 1501 && ttsb.Iterate <=
        1700;
select sum(Aggregate) from ttsb where ttsb.Iterate >= 1100 && ttsb.Iterate <= 1300;
myAggregate = ttsb.Aggregate;

select sum(Change) from ttsb where ttsb.Iterate >= 1301 && ttsb.Iterate <= 1500;
my2ndAggregate = ttsb.Change;

select sum(Aggregate) from ttsb where ttsb.Iterate >= 1501 && ttsb.Iterate <= 1700;
myAggregate += ttsb.Aggregate;

}
```

Средства мониторинга производительности

Без средств мониторинга исполнения реализованной прикладной логики вся новая функциональность с точки зрения вопросов производительности писалась бы практически вслепую. К счастью, среда разработки Microsoft Dynamics AX содержит набор легких в использовании средств, которые выполняют мониторинг клиент/серверных вызовов, обращений к базе данных и исполняемой прикладной логики. Данные средства обеспечивают вас достаточно полной информацией об исследуемой функциональности. Получаемая информация интегрируется прямо в среду разработки, делая возможным прямой переход к соответствующему коду X++.

Trace Parser в Microsoft Dynamics AX

Microsoft Dynamics AX Trace Parser – это инструмент с графическим пользовательским интерфейсом, основанный на использовании Microsoft SQL Server и инфраструктуры Трассировки событий для Microsoft Windows

(ETW). Microsoft Dynamics AX TraceParser был значительно улучшен в Microsoft Dynamics AX 2012, были добавлены новые возможности, повышена производительность и удобство использования. Падение производительности системы при выполнении трассировки сравнительно невелико, ETW создает дополнительную нагрузку на систему порядка 4%.

Включать трассировку могут только пользователи с правами администратора, пользователи из группы Пользователи системного монитора (Performance Log Users), а также службы, выполняющиеся под учетными записями *LocalSystem*, *LocalService* и *NetworkService*. Для использования Панели трассировки в клиенте Microsoft Dynamics AX пользователь должен входить в группу локальных администраторов или в группу Пользователи системного монитора. То же самое требование относится к пользователям Системного монитора Windows. Кроме того, у пользователя должны быть права на запись в каталог, где хранятся файлы с результатами трассировки.

Trace Parser обеспечивает быстрый анализ собранных данных в случае поиска самых долго выполняющихся участков кода, самых долго выполняющихся запросов SQL, большого количества вызовов, а также других метрик, полезных при отладке проблем производительности. Более того, он предоставляет доступ к дереву вызовов для всего исполняемого кода, что позволяет быстро понять структуру незнакомого кода. Также при выполнении анализа возможно с помощью поиска переходить к дереву вызовов для определения мест вызова проблемного кода.

Trace Parser включен в поставку Microsoft Dynamics AX 2012, а также бесплатно доступен для скачивания на Partner Source и Customer Source. Чтобы установить Trace Parser, запустите программу установки Microsoft Dynamics AX 2012 и в ней выберите Add or Modify Components > Developer Tools > Trace Parser.

Новые возможности Trace Parser

В Microsoft Dynamics AX 2012 включено несколько новых возможностей для Trace Parser, которые помогут вам быстро выявить причину проблемы с производительностью.

- **Мониторинг вызовов методов.** Если вы щелкните правой кнопкой мыши по строке в представлении X++/RPC и затем выберите Jump To Non-Aggregated View, то можете увидеть такую информацию, как все ли вызовы метода длились одинаковое время, или же какой-то вызов выбивается из общего ряда. Эта же функция доступна в представлении SQL.

- **Мониторинг клиентских сессий.** Если в представлении Non-Aggregated или Call Tree между клиентом и сервером есть RPC-вызов, то можете щелкнуть правой кнопкой мыши по строке вызова и затем выбрать Drill Through To Client Session (перейти к клиентской сессии).
- **Переход между представлениями.** Если вы хотите перейти из представления Call Tree к представлению Non Aggregated X++/RPC, то можете щелкнуть правой кнопкой мыши и выбрать соответствующий пункт контекстного меню.
- **Мониторинг событий.** В представлении Non Aggregated X++/RPC или же Call Tree вы можете выбрать два или более события, удерживая нажатой клавишу Ctrl, затем щелкнуть правой кнопкой мыши и выбрать Show Time Durations Between Events (показать время, прошедшее между событиями). Это чрезвычайно полезно для анализа и решения проблем с асинхронными событиями.
- **Просмотр подробных сведений о таблице.** Вы можете выбрать пункт меню View > Table Details для просмотра подробных сведений о таблице из Microsoft Dynamics AX. Для этой функции необходим Business Connector .NET, так же как и для просмотра исходного кода X++.
- **Сравнение трассировок.** Вы можете выбрать пункт меню View > Trace Comparison, и в результате откроется форма, на которой вы можете сравнить две трассировки.

Перед началом трассировки

Перед тем как запускать трассировку, хотя бы один раз прогоните процесс, который собираетесь трассировать, чтобы в трассировку не попала информация о загрузке метаданных. Это называется *трассировкой в прогретом состоянии* и является рекомендуемым подходом, поскольку позволяет вам сконцентрироваться на реальной проблеме с производительностью, а не на загрузке и кэшировании метаданных. Затем вы можете выполнить подготовительные работы, чтобы насколько возможно сократить время между началом трассировки и выполнением того процесса, который хотите трассировать.

В Microsoft Dynamics AX 2009 вам приходилось настраивать трассировку в нескольких различных местах; в Microsoft Dynamics AX 2012 есть лишь три места для настройки параметров трассировки. Кроме того, теперь используется один файл трассировки как для клиента, так и для сервера. Запустить трассировку вы можете одним из трех способов.

- Из Панели трассировки в клиенте Microsoft Dynamics AX 2012.
- Из Системного монитора Windows.
- С помощью инструментальных средств в коде.

Каждый из способов детально описан в следующих разделах.

Запуск трассировки из клиента

Как упоминалось ранее, для использования Панели трассировки вы должны войти под учетной записью администратора. Возможности, доступные в Панели трассировки, описаны в табл. 13-2.

Табл. 13–2. Параметры на Панели трассировки

Элемент	Описание
Запуск трассировки	Запустить трассировку после того, как вы укажете, где хотите сохранить файл трассировки
Остановка трассировки	Остановить трассировку и завершить запись информации в файл трассировки
Отменить трассировку	Остановить трассировку без сохранения информации в файл трассировки
Открыть трассировку	Открыть файл трассировки в Trace Parser
Сбор трассировки сервера	Собирать данные как клиента, так и сервера
Круговая регистрация	Укажите размер файла и собирайте информацию, пока не нажмете Остановка трассировки. При выборе этого параметра данные перезаписываются, поэтому в файле вы получите наиболее последние данные. Этот параметр появился в Microsoft Dynamics AX 2012 и особенно полезен, если вы хотите трассировать процессы, которые выполняются дольше, скажем, 10 минут. Вы можете использовать эту настройку для записи трассировки в середине работы долго выполняющегося процесса
Связать параметры	Позволяет пользователям получить реальные значения, передаваемые базе данных, вместо параметризированных запросов. По умолчанию эта настройка выключена, потому что потенциально она может привести к сбору конфиденциальной информации
Детальная база данных	Собирать информацию о числе выбранных записей и времени, затраченном на их выборку

Табл. 13–2. Параметры в Панели трассировки (*окончание*)

Элемент	Описание
RPC	Собирать информацию о числе совершенных RPC-вызовов
SQL	Собирать SQL-запросы, которые AOS отправляет базе данных
TraceInfo	Показывать информацию о том, какой процесс привел к записи события
TTS	Логировать операторы *ttsBegin*, *ttsCommit* и *ttsAbort*
XPP	Логировать выполняемые вызовы X++
Маркер XPP	Копировать маркеры, которые добавляются во время трассировки, в файл трассировки
Доступ клиента	Собирать информацию о том, какие формы открывались и закрывались и какие нажимались кнопки на формах
Информация о параметрах XPP	Собирать информацию о параметрах, передаваемых методам X++. По умолчанию эта настройка выключена, потому что потенциально она может привести к сбору конфиденциальной информации

Для запуска трассировки из Панели трассировки выполните следующее.

1. В клиенте Microsoft Dynamics AX 2012 откройте рабочую область разработки, нажав Ctrl+Shift+W.
2. Выберите пункт меню Сервис > Панель трассировки (рис. 13-11).
3. Настройте параметры трассировки. Например, если вы хотите собрать информацию только о работе клиента, то сбросьте флажок Сбор трассировки сервера.
4. Приведите ваш процесс в разогретое состояние (как описано ранее) и затем нажмите Запуск трассировки.
5. Выберите, где должен быть сохранен файл трассировки.
6. Выполните ваш процесс и затем нажмите Остановка трассировки.
7. Нажмите Открыть трассировку, чтобы открыть полученный файл трассировки в Trace Parser.

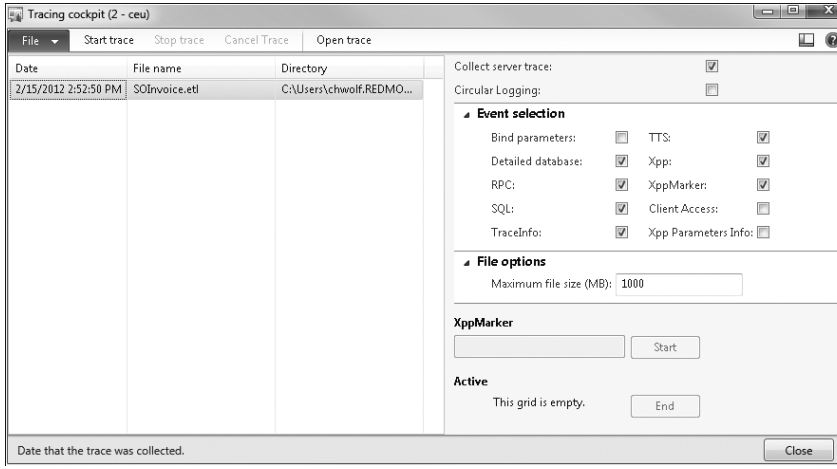

Рис. 13–11. Панель трассировки

Запуск трассировки из Системного монитора Windows

Для запуска трассировки из Системного монитора Windows проделайте следующее.

1. Нажмите Пуск > Выполнить и введите **perfmon**.
2. Разверните группы сборщиков данных.
3. Щелкните правой кнопкой мыши по Особый и выберите Создать > Группа сборщиков данных.
4. Выберите Создать вручную и нажмите Далее.
5. Выберите Данные отслеживания событий и нажмите Далее.
6. Рядом с поставщиками нажмите Добавить и в списке Поставщики отслеживания событий выберите Microsoft-DynamicsAX-Tracing, после чего нажмите OK.

Примечание. Если вы используете Системный монитор Windows, то по умолчанию трассируются все события, включая те, что могут собирать конфиденциальную информацию. Для предотвращения этого щелкните Изменить и выберите события, которые вас интересуют. В описании событий, которые могут собирать конфиденциальную информацию, есть соответствующее замечание.

7. Нажмите Далее и затем запомните корневой каталог, где будут сохраняться ваши трассировки.
8. Нажмите Далее, чтобы изменить пользователя, выполняющего трассировку, на администратора, и нажмите Готово.
9. На правой панели Системного монитора Windows щелкните правой кнопкой мыши по созданной группе сборщиков данных и выберите Свойства.
10. В окне свойств щелкните по вкладке Буферы трассировки и измените значения параметров по умолчанию.

    Настройки по умолчанию для буферов не подходят для сбора событий трассировки Microsoft Dynamics AX, потому что тут в небольшой промежуток времени может генерироваться большое количество событий, и буферы слишком быстро заполнятся. Измените настройки, как указано ниже, а прочие настройки оставьте по умолчанию.

    - Размер буфера: 512 KB.
    - Минимум буферов: 60.
    - Максимум буферов: 60.
11. Чтобы начать трассировку, щелкните по набору собираемых данных в левой панели и нажмите Запуск.

Запуск трассировки с помощью инструментальных средств в коде

Для запуска и остановки трассировки вы можете использовать класс *xClassTrace* из Панели трассировки. Ниже приводится пример трассировки разноски документа по заказу на продажу (\Classes\SalesFormLetter).

```
// Добавьте
xClassTrace xCt = new xClassTrace();

// в объявление переменных.
// ...код...

    if (salesFormLetter.prompt())
    {
        xClassTrace::start("c:\\temp\\test1.etl");
        xClassTrace::logMessage("test1");
        xCt.beginMarker("marker"); // добавляет маркер в определенной точке
                                   // трассировки, чтобы улучшить ее читаемость. Вы
                                   // можете добавлять несколько маркеров в одну
                                   // трассировку
```

```
            salesFormLetter.run();

            xCt.endMarker("marker");
            xClassTrace::stop();

            outputContract = salesFormLetter.getOutputContract();
            numberOfRecords = outputContract.parmNumberOfOrdersPosted();
        }
    // ...код...
```

При вызове *xClassTrace::start* вы можете указать различные параметры, управляющие тем, какие события необходимо трассировать, использовать ли круговую регистрацию и т.д. Чтобы узнать, какое ключевое слово связано с каким параметром, поставьте точку останова в методе класса *SysTraceCockpitcontroller\startTracing* и затем запускайте трассировку из Панели трассировки, выбирая различные события.

Импорт трассировки

Для импорта трассировки откройте Microsoft Dynamics AX Trace Parser и нажмите Import Trace (для импорта файла трассировки вы также можете использовать форму открытия трассировки). За один раз можно импортировать несколько файлов трассировки.

Анализ трассировки

После загрузки файлов трассировки в Trace Parser вы можете анализировать их с помощью встроенных представлений. Когда вы открываете трассировку из вкладки Обзор, вы видите итоговую информацию, дающую представление о том, где в рамках трассировки было потрачено больше всего времени. Выберите сессию на вкладке Обзор; если вы сами записывали трассировку, то выберите свою сессию, если же вы получили файл трассировки от кого-то другого, то выберите сессию того, кто записывал трассировку. После выбора сессии вы увидите окно с информацией, похожей на рис. 13-12, но только для этой сессии. Чтобы вернуться к итогам для всех сессий, поставьте галочку Show Summary Across All Sessions (показывать итоги для всех сессий).

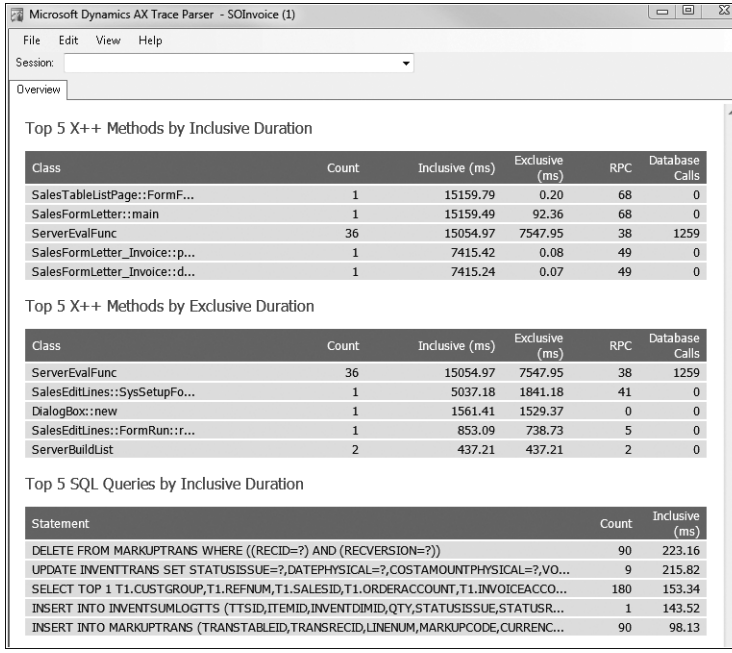

Рис. 13–12. Обзор трассировки

После выбора сессии из выпадающего списка вы можете просматривать и искать информацию среди вызовов методов X++, RPC-вызовов или запросов SQL, либо можете просмотреть дерево вызовов сессии. Лучше всего сначала попытаться найти места, где можно достичь быстрых улучшений, для чего отсортируйте данные по общей исключающей длительности. Затем разбивайте процесс на фрагменты с целью более тонкой настройки, для чего сортируйте их по общей включающей длительности. Вы можете перейти к представлению Call Tree из методов X++, RPC-вызовов и из представления SQL. **Используйте представление X++/RPC для понимания** шаблонов, присутствующих в вашей трассировке, как показано на рис. 13-13.

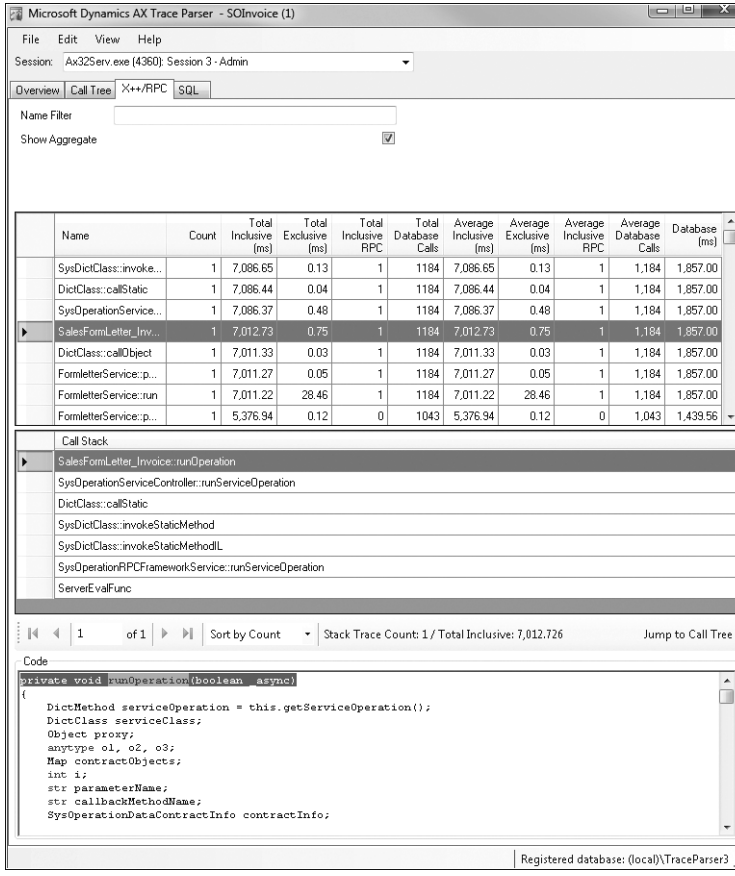

Рис. 13–13. Представление X++/RPC

Представление SQL **(см. рис. 13-14) дает вам возможность быстро узнать,** какие запросы выполнялись и сколько времени заняло их выполнение, а также выборка данных.

Примечание. Время выполнения запроса и время выборки данных измеряются отдельно.

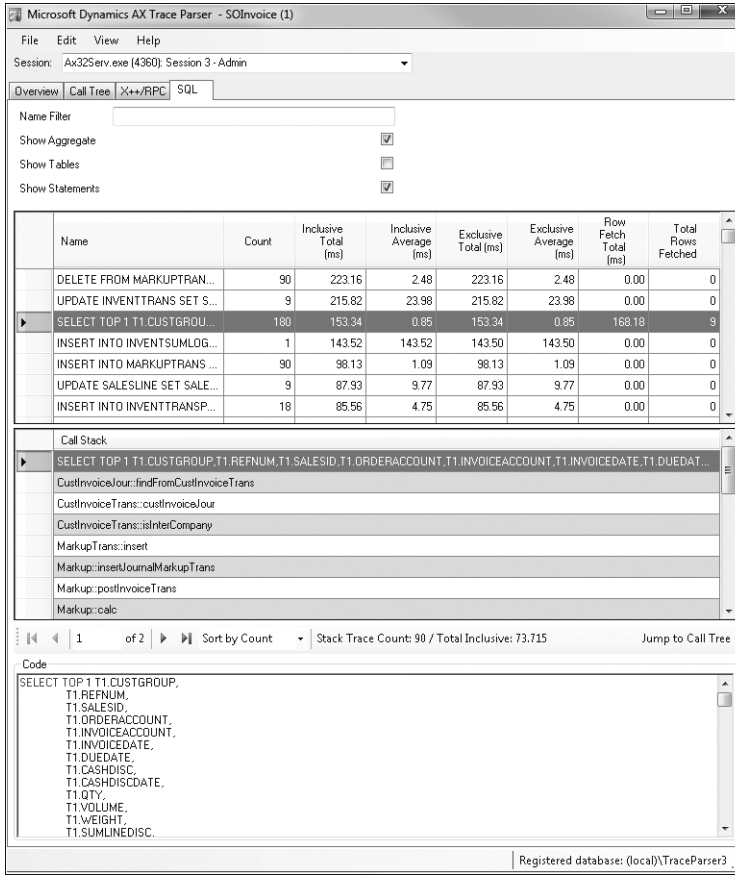

Рис. 13–14. Представление SQL

Представление Call Tree (см. рис. 13-15) особенно полезно при определении циклов и других шаблонов, на выполнение которых затрачивается значительное время.

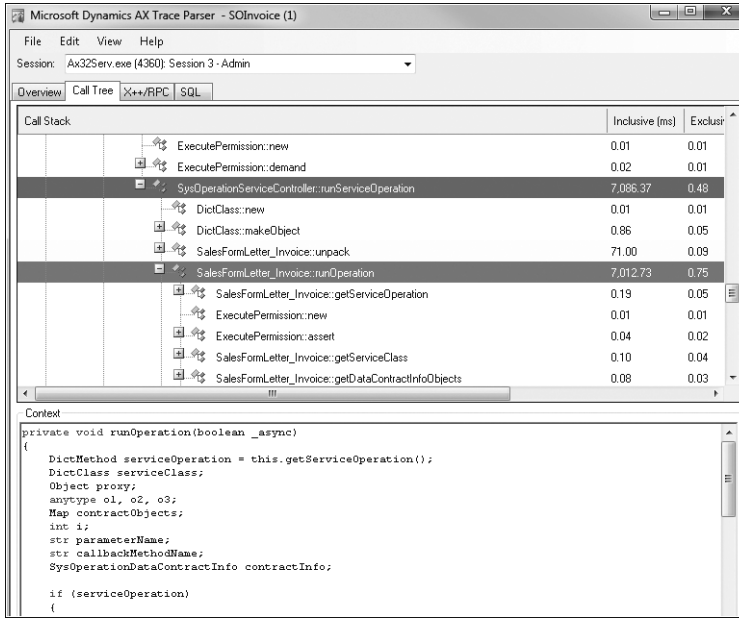

Рис. 13–15. Представление Call Tree

Решение проблем с трассировкой

В этом разделе представлена информация о способах решения некоторых из наиболее часто встречающихся проблем с трассировкой.

Трассировка не запускается. Если трассировка не запускается, убедитесь, что пользователь, который ее запускает, является членом локальной группы администраторов или группы Пользователи системного монитора (Performance Log Users).

Трассировка вызывает проблемы с производительностью. Если вы запускаете трассировку с клиента на хосте, на котором работает AOS, то получите один файл трассировки, если же с другого хоста, то вы получите два файла: один на хосте клиента и один на хосте сервера AOS. При запуске одновременно более чем одной трассировки работа системы замедлится, потому что в этом случае трассировка будет требовать много вычислительных ресурсов и дискового пространства. Рекомендуется не запускать трассировку на сервере AOS, **на котором выполняется множество** клиентских сессий.

Трассировка не дает осмысленных данных. Если код X++ выполняется как CIL, то результаты трассировки могут быть бессмысленными. В табл. 13-3 перечислены сценарии, из-за которых могут возникнуть проблемы с трассировкой, а также рекомендации для каждого случая.

Табл. 13–3. Решение проблем с трассировкой кода X++, выполняющегося в CIL

Сценарий	Рекомендации
Код X++ запускается в CIL с помощью *RunAs*	В рабочей области разработки выберите Сервис > Параметры и на вкладке Разработка снимите флажок Выполнять бизнес-операции в CIL
Сервисы, вызываемые извне Microsoft Dynamics AX, или сервисы из группы AxClient	Зачастую проще всего написать тестовый job или класс для вызова сервиса из Microsoft Dynamics AX. Если по каким-либо причинам этот вариант не подходит, то используйте для трассировки сервиса профайлер из Microsoft Visual Studio
Пакетные задания выполняются в CIL	Выполните код вне инфраструктуры пакетных заданий. Постарайтесь сократить длительность операции, например, ограничив операцию небольшим числом задач, которые могут быть обработаны за несколько минут. Если это невозможно, используйте профайлер из Visual Studio, описанный в конце этой главы

Мониторинг обращений к базе данных

При разработке и тестировании прикладной логики вы также можете отслеживать обращения к базе данных. Отслеживание операций базы данных может быть включено на вкладке SQL формы Параметры (меню Сервис > Параметры). Можно отслеживать как все исполняемые SQL-операторы, так и только длинные запросы, предупреждения и взаимоблокировки. Операторы SQL могут выводиться в Infolog, окно сообщений, таблицу базы данных или же в файл на жестком диске. Если операторы передаются в окно Infolog, то можно использовать контекстное меню для открытия выполненного оператора БД в форме Трассировка запросов SQL, где легко можно просмотреть весь оператор базы данных, а также путь к методу, который его выполнил.

Примечание. Этой возможностью следует пользоваться только для долговременного мониторинга длинных запросов, но даже в этом случае данную возможность следует использовать с осто-

рожностью, поскольку она создает дополнительную нагрузку на систему.

На форме Трассировка запросов SQL вы можете скопировать запрос и при наличии установленной SQL Server Management Studio (SSMS) вставить запрос в новое окно запросов SSMS. Если среда времени выполнения Microsoft Dynamics AX использует параметризированные запросы, то значения параметров будут отображаться в запросе в виде вопросительных знаков. Вы должны заменить их переменными или константами, прежде чем запрос можно будет передать в SQL Server Query Analyzer. Если же среда времени выполнения использует литералы, то запрос можно передать в SQL Server Query Analyzer и выполнить в исходном виде.

При трассировке запросов SQL в Microsoft Dynamics AX среда времени выполнения отображает только DML-запросы. Она не отображает другие команды, отсылаемые базе данных, такие как завершение транзакции или изменение уровня изоляции. С версии SQL Server 2008 и выше вы можете использовать SQL Server Profiler для отслеживания таких запросов с использованием классов событий *RPC:Completed* и *SP:StmtCompleted* в категории Stored Procedures, а также события *SQL:BatchCompleted* в категории TSQL, как показано на рис. 13-16.

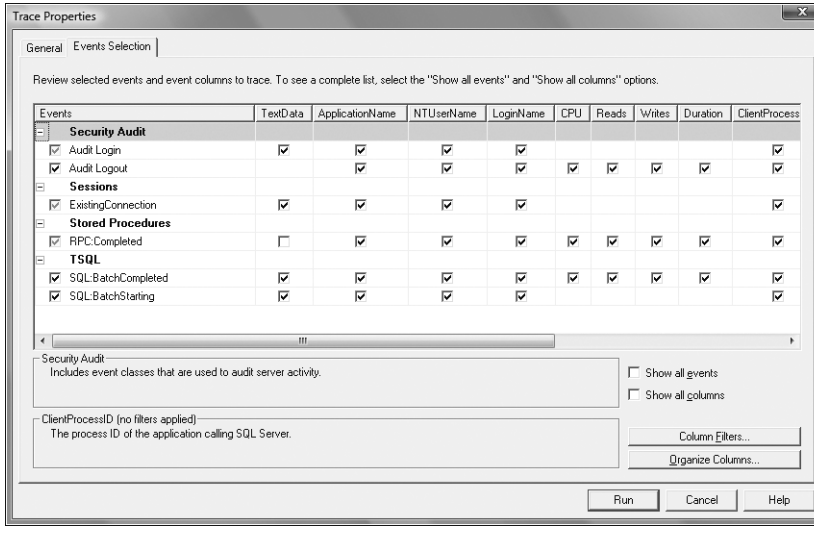

Рис. 13–16. Трассировка событий в SQL Server Profiler

Использование контекста соединения SQL Server для поиска SPID или пользователя, связанного с клиентской сессией

Вы можете использовать Server Process ID (SPID) или код пользователя, связанного с клиентской сессией, для решения широкого круга проблем, таких как соперничество за ресурсы или медленно выполняющиеся запросы. В прежних версиях Microsoft Dynamics AX форма Активных пользователей содержала колонку для SPID клиентской сессии. В Microsoft Dynamics AX 2012 информация о пользовательских сессиях может быть включена в контекст соединения SQL Server. Добавление этой информации создает небольшую дополнительную нагрузку с точки зрения производительности. Более подробную информацию вы можете найти в публикации «Finding User Sessions from SPID in Dynamics AX 2012» в блоге Thoughts on Microsoft Dynamics AX (*http://blogs.msdn.com/b/amitkulkarni/archive/2011/08/10/finding-user-sessions-from-spid-in-dynamics-ax-2012.aspx*).

После применения информации из указанной публикации вы также сможете использовать информацию о сессии в выборке, возвращаемой приведенным ниже запросом, включая коды пользователей Microsoft Dynamics AX и до некоторой степени запросы, которые они в данный момент выполняют:

```
select top 20 cast(s.context_info as varchar(128)) as ci,text,query_plan,* from
sys.dm_exec_cursors(0) as ec cross apply sys.dm_exec_sql_text(sql_handle) sql_text,
sys.dm_exec_query_stats as qs cross apply sys.dm_exec_query_plan(plan_handle) as
plan_text,sys.dm_exec_sessions s
where ec.sql_handle = qs.sql_handle and ec.session_id = s.session_id order by ec.worker_time
desc
```

Журнал активности пользователей

Вы можете использовать журнал активности пользователей для отслеживания активности множества пользователей во время выполнения ими каждодневной работы. Данные журнала пишутся в таблицу SysClientAccessLog. Более подробную информацию об этой функциональности вы можете найти в публикации «Client Access Log» в блоге команды Microsoft Dynamics AX, занимающейся вопросами производительности (*http://blogs.msdn.com/b/axperf/archive/2011/10/14/client-access-log-dynamics-ax-2012.aspx*).

Профайлер Visual Studio

Как упоминалось ранее, для некоторых процессов, пожалуй, единственным способом трассировки является использование профайлера Visual Studio. Ниже в общих чертах описано, как использовать профайлер Visual Studio вместе с Microsoft Dynamics AX.

Примечание. Профайлер Visual Studio доступен в редакциях Visual Studio 2010 Premium и Visual Studio 2010 Ultimate.

1. В Visual Studio выберите пункт меню Debug > Options And Settings.
2. В левой панели формы Options щелкните Debugging, затем Symbols и убедитесь, что файл символов загружен для каталога XppIL того сервера AOS, на котором вы хотите выполнить профилировку. (Инструменты профилировки используют файлы символов [.pdb] для получения символических имен, таких как названия функций в бинарных файлах программы.)
3. Выберите пункт меню Analyze > Launch Performance Wizard для создания новой сессии анализа производительности.
4. Примите настройки по умолчанию для частоты квантования CPU и выберите AOS, на котором вы хотите выполнять профилировку, но не начинайте ее сразу.
5. Откройте Performance Explorer, щелкните правой кнопкой по верхнему узлу вашей сессии (рис. 13-17) и откройте окно свойств.

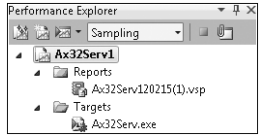

Рис. 13–17. Performance Explorer

6. В окне свойств перейдите к настройке Sampling и уменьшите интервал квантования до 100000 или 1000000, чтобы получить более точные результаты.
7. Подготовьте выполнение действия, которое вы хотите трассировать, и затем щелкните по Attach/Detach, чтобы подключиться к процессу AOS.
8. После завершения профилировки щелкните Attach/Detach, чтобы отключиться от процесса AOS.

Важно. Не нажимайте кнопку Stop Profiling – это приведет к тому, что AOS перестанет отвечать.

После завершения профилировки Visual Studio формирует отчет, который поможет вам понять, где кроются проблемы с производительностью, как показано на рис. 13-18.

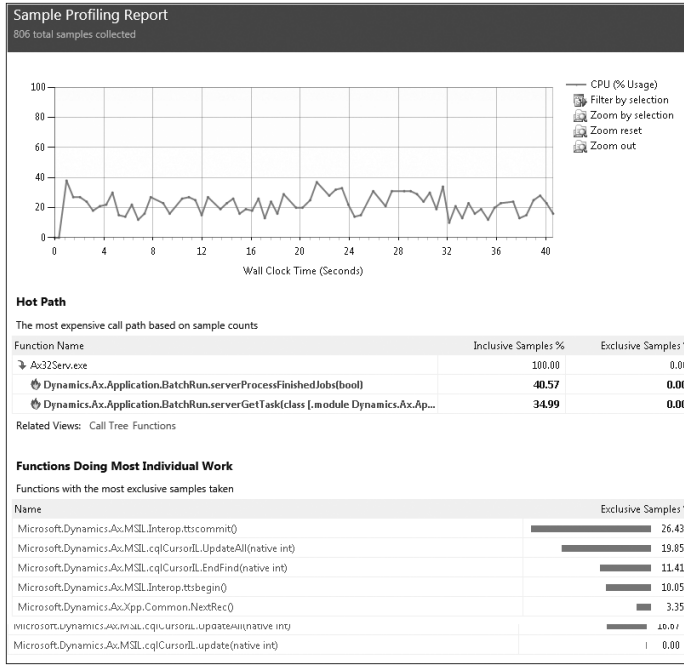

Рис. 13–18. Отчет профилировки

Отчет предлагает несколько представлений, таких как Summary, Call Tree и Functions, и позволяет просмотреть функции, которые вызвали ту функцию, что вы анализируете в данный момент. Если вы установили Инструменты Visual Studio для Microsoft Dynamics AX, то также можете быстро, не покидая Visual Studio, перейти к коду X++ тех методов, которые упоминаются в отчете.

Совет. Чем меньше интервал квантования, тем лучше качество профилирования, но и тем больше собирается данных.