Ретроспективная основа совместной реорганизации сложных информационных ресурсов§

© С.В. Знаменский

Институт программных систем РАН имени А.К. Айламазяна
svz@latex.pereslavl.ru

Аннотация

Информационные ресурсы, адресованные комплексным проблемам, непрерывно дорабатываются с привлечением широкого круга специалистов и организаций.
Такая доработка часто нуждается в неожиданных коррекциях структуры ресурса, параллельной разработке и сопоставительном тестировании пользователями нескольких версий его составляющих, удалении лишнего и восстановлении ранее доступного.
Обсуждается идея построить представление данных в такой системе на неизменяемых записях, адресуемых с учётом даты-времени и авторства их создания и обеспечивающей ускорение многопользовательской доработки за счёт

- непосредственного доступа к прежним состояниям информации и существенным особенностям её изменения;
- локализации ошибок исполнения во времени и информационном пространстве;
- многослойного представления реорганизуемых данных (с разными комбинациями активных слоёв для разных пользователей) и многоуровневого тестирования.

1 Введение

Часть информационной системы, обладающую в каждый момент времени состоянием, полостью согласованном с её бизнес-логикой, принято называть компонентой. Кроме данных, относящихся к состоянию, компонента может иметь данные незавершённых транзакций, каждая из которых скачком должна перевести ей в новое жёстко согласованное состояние. Компонента может поддерживаться либо реляционной СУБД, либо иными средствами.

Несмотря на огромное количество теоретических и прикладных разработок, направленных на повышение эффективности механизмов обработки транзакций, компонента может гарантировать ожидаемо быструю реакцию на каждый запрос к системе только в случае прозрачной фиксированную логики без трудоёмких алгоритмов обработки данных. Иначе очередной скачок к новому состоянию рано или поздно потребует неожиданно много времени.

По стандартной технологии (например, СОА), сложные информационные системы разбиваются на компоненты, обменивающиеся системными сообщениями.

Компонентная архитектура эклектична. Наличие консистентного состояния в каждой компоненте резко контрастирует с очень вероятной рассогласованностью данных из разных компонент. При наличии трудоёмких обработок системных сообщений это приходится предусматривать и учитывать в протоколах взаимодействия компонент.

1.1 Цель исследований

Целью работы является преодоление упомянутой эклектичности для систем, направленных на высококачественную информационную поддержку такой совместной творческой деятельности, которая включает в себя совершенствование организации самой этой деятельности. Подобные устойчиво развивающиеся системы сейчас базируются на СОА и настолько ресурсоёмки и дороги в разработке и сопровождении, что доступны лишь десяткам богатейших организаций мира. Причина в том, что отсутствие однонаправленности (определяющей быстроту реакции поисковых сервисов интернет) и изменчивость структур управления ланными делают задачу эффективного разделения сложной требуемой бизнес-логики на компоненты разрешимой лишь под постоянным контролем крупной команды высококвалифицированных специалистов -разработчиков.

Потребность в устойчиво развивающихся информационных системах высока не только в богатейших организациях.

Обсуждаемая в работе гипотеза состоит в том, что эту потребность мог бы при достаточно скромной стоимости разработки и владения удовлетворить нестандартный подход к созданию устойчиво развивающихся ИС.

Главной целью разработки ставятся задачи организации более сложной совместной деятельности по информационному обслуживанию,

Труды 13[й] Всероссийской научной конференции «Электронные библиотеки: перспективные методы и технологии, электронные коллекции» - RCDL'2011, Воронеж, Россия, 2011.

в которой пользователи и группы с не очень высоким уровнем квалификации смогут с минимальным риском самостоятельно произвольно перестраивать бизнес-логику взаимодействия в доверенных им частях сложной информационной системы.

Такая постановка обусловлена общеизвестными крайне неутешительными прогнозами обострения дефицита программистов высокой квалификации.

1.2 Идея подхода

Ключевой идеей является замена механизма системных сообщений между компонентами обращениями к единой единой ретроспективной СУБД.

Ретроспективной будем называть СУБД, в которой

- Каждая запись фиксируется с указанием момента записи, а, если запись возникла в результате обработки ранее сделанных записей, то и с указанием момента, на который актуальны исходные данные.
- Доступ на чтение предоставляется только к ранее записанным данным;
- Доступные на чтение записи неизменны вплоть до физического удаления из системы, осуществляемого автоматически через большой срок после логического удаления.

В отличие от битемпоральных реляционных СУБД, ретроспективная не берёт на себя задачу обеспечить соответствия состояния данных бизнес-логике приложения. Она обеспечивает только сохранность полной истории изменения данных и эффективный доступ к ранее сохранённой информации.

Это частично вытесняет проблему конкурентности доступа с уровня реализации бизнес-логики частью на уровень самой бизнес логики (при наличии реальной основы для конкурентности). Если, например, несколько пользователей могут претендовать на уникальный неделимый ресурс (последний билет на самолёт), то их запросы фиксируется и, например, через десятую долю секунды обрабатываются. Бизнес-логика должна определить, как выбирается победитель в случае практически одновременной подачи заявки — стандартно по времени и ID или с учётом каких-то привилегий.

Другой пример согласованности — это постоянство суммы средств на счетах в процессе сделки. Это элемент упрощенной для реляционной СУБД математической модели, имеющий слабое отношение к реальной жизни. В жизни сумма доступных обладателям денег на счетах ненадолго сокращается в период перевода денег.

Вынос логической основы для конкурентности на уровень бизнес-логики не только приближает модель к реальности, но и снимает с СУБД тяжелейшее требование во что бы то ни стало обеспечивать полную согласованность.

ретроспективная СУБД должна обрабатывать запросы на чтение информации так, чтобы в процессе применения изменений выдавать прежнее состояние данных на любой промежуток времени. При выдаче данного используется последняя из записей этого данного, помеченная моментами времени, предшествующими соответствующим концам указанного промежутка.

Такой подход открывает непривычную возможность сочетания живых интерактивных пользовательских интерфейсов со сложной фоновой обработкой и упрощает отладку и расследование инцидентов.

Образно говоря, нажимая кнопку или переходя по ссылкам пользователь

- никогда не увидит сообщения о занятости системы вместо ожидавшейся странички;
- мгновенно увидит результат перехода по ссылке (иногда с предупреждением о том, что данные актуальны, например, на вчерашний вечер, которое быстро само исчезнет с обновлением либо сменится предложением перезагрузить страницу);
- при нажатии кнопки так же быстро увидит, что нажатие принято и сможет продолжить работу, не дожидаясь результатов обработки;

Непривычность такой частично упреждающей фоновой обработки оправдана снижением на порядки суммарного времени ожидания пользователями реакции сложной системы без привлечения дополнительных вычислительных ресурсов. При многократных перегрузках системы будет расти время ожидания, но обслуживание не прекратится.

На ретроспективной основе могут на уровне более низком, чем бизнес-логика быть реализованы и дополнительные возможности:

- благодаря локализации вычислений, параллельная отладка изменившейся бизнес-логики на живых данных может проводиться незаметно для других пользователей системы;
- смягчение требований к синхронизации позволяет кардинально повысить катастрофоустойчивость дублированием информации на удалённых параллельно работающих серверах;
- прежний доступ пользователя к прежней информации системы может быть обеспечен на случай временной поломки бизнес-логики приложения;

2 Область применимости

Чтобы пояснить необходимость рассмотрения нестандартной технологии, рассмотрим простейшую из решаемых им задач —

классификация большого множества ресурсов. Независимо от того, какие ресурсы классифицируются — публикации, советы по устранению проблем с ПО, описания биологических объектов, товаров или образовательные ресурсы, можно считать, что каждый объект представлен некоторым url и рассмотреть задачу построения каталога веб-ресурсов.

2.1 Коллаборативный каталог-классификатор

Обычно большие каталоги-классификаторы строятся и поддерживаются небольшими командами специалистов и содержат до нескольких сотен тысяч ресурсов. Готовые классификаторы, как правило, нуждается в уточнении, расширении (часто на порядки) и в согласовании с другими.

Возникает потребность привлечь к классификации всех заинтересованных специалистов (либо, вообще, всех желающих), предоставив каждому из них возможность быстро, удобно, эффективно и конструктивно классифицировать ресурсы, исправлять ошибки, соглашаться с общим мнением или оспаривать его.

Новичкам такая система выдаст каталог, оптимизированный с учётом всех мнений всех пользователей-разработчиков, а пользователь, внёсший изменения, увидит свои изменения учтёнными на фоне общих изменений.

При наличии хорошего начального наполнения и привлечения достаточного круга пользователей, новых пользователей будут привлекать простота добавления в каталог и изменения его структуры, богатство и оптимальность каталога.

Не исключено, что такой каталог мог бы на равных конкурировать с поисковыми сервисами интернет, поскольку пройти по каталогу проработанной структуры быстрее и легче, чем правильно сформулировать запрос, а в результате получается не список предположительно релевантных ресурсов, а список ресурсов, рекомендованных другими пользователями в данном конкетсте.

Для такой ИС критичными являются: дружелюбный пользовательский интерфейс, масштабируемость и надежность, в частности, защищённость от спама и других угроз. Чем шире круг участников, тем полнее, актуальнее и качественнее будет каталог при правильной организации ИС.

2.2 Описание функциональности каталога

Идея дружелюбного пользовательского интерфейса нетривиальна, но вполне обозрима. Есть множество U пользователей, пополняемое пользователями множество R ресурсов и также пополняемое пользователями множество T имён признаков (они же имена подкаталогов). Кликая по имени не выбранного признака, пользователь тем самым добавляет его в текущий контекст $C \subset T$, попадая на страничку нового контекста, на которой между выбранными и доступными для выбора признаками расположены ресурсы в рейтинговом порядке.

Ниже выбранных признаков следуют другие характерные для всех ресурсов этого контекста признаки.

Пользователь одним движением мышкой может

- изменить и зафиксировать рейтинг и контекст ресурса;
- изменить порядок предпочтения признаков;
- переместить ресурс в полкаталог либо вынести в родительский каталог;
- добавить или удалить ресурс, или каталог в данном контексте;
- переименовать или скопировать подкаталог;
- вынести подкаталог наружу, изменив имя.

Все эти действия фиксируют индивидуальные рейтинги ресурсов и каталогов (для подкаталогов контекстно), доопределяют и изменяют пользовательскую функцию инцидентности

$I_u: R \times T \to \{0,1\}$

Авторитетность A_u пользователя в контексте рассчитывается по относительному количеству других пользователей, поддержавших его выбор в этом и ближайших контекстах.

Значения функции инцидентности усредняются по U с весом авторитетности, образуя идеальную функцию инцидентности I,. С такими же весами усредняются рейтинги. Полученные данные используются для формулировки задачи оптимизации, решением которой является структура каталога, максимально отвечающая пожеланиям пользователей.

При такой оптимизации в корневом разделе со временем могут оказаться совсем иные признаки, то есть структура каталогов сможет беспрепятственно перестраиваться следуя изменяющимся потребностям пользователей.

Алгоритмы и настройки оптимизации каталога и расчёта авторитетности пользователя, дизайн контекстной страницы, должны совершенствоваться с ростом каталога.

Должна развиваться система подсказок, помогающих пользователям упрощать каталог, выявляя равнозначные признаки или наборы признаков.

Пользователю удобно иметь возможность увидеть произошедшие за указанный им период изменения в контексте и пересмотреть свои действия (или действия, сделанные от его имени).

Потребуется добавить платные сервисы для самоокупаемости. Если мнение пользователя отличается от других, ему должно быть это ненавязчиво показано с возможностью простого обсуждения наличия конкретного признака у конкретного ресурса или рейтинга конкретного ресурса или каталога.

Очевидно, что каждый контекст при

оптимизации может улучшаться практически независимо от других контекстов. Контексты, к которым утрачен интерес, можно не оптимизировать, а популярные контексты необходимо оптимизировать чаше и, может быть, более тщательно.

Сложность этой задачи в сочетании необходимости многих сложных изменяющихся обработок с быстрой реакцией на действия пользователя при умеренной продолжительности обработки изменений. Она препятствует использованию стандартной основы.

2.3 Другие возможные применения

Приведённый пример далеко не является исключением. В интернете наблюдается качественный и количественный рост

- ресурсов открытых технических стандартов, включая библиографические классификаторы;
- ресурсов открытых образовательных стандартов и систем дистанционного образования;
- общедоступных ресурсов по биологии и другим наукам, склонным к пересмотру концепций и классификаций;
- ресурсов открытого исходного кода больших программных систем;
- государственных и корпоративных ресурсов законодательно-нормативной информации.

Потребность в частой коррекции структуры подобного ресурса обусловлена не столько неполнотой предпроектного обследования, сколько

- наличием внутренних несогласованностей и требующих исправления противоречий,
- рассогласованностью политик и приоритетов различных пользователей, создавших ресурс,
- непредсказуемостью изменений в предметной области.

Рост масштабов, сложности и глобальности информационных систем актуализирует проблему совместной доработки их наполнения.

Это наполнение сложнее, чем приведённый пример. Для его качественного улучшения нужны разнообразные новые инструменты, часто узко специальной направленности. Такие инструменты зачастую могут быть созданы или адаптированы только самими пользователями и это процесс полезно организовать в самой системе, .

Непрерывная интеграция [1] обещает удешевление и ускорение разработки подобного перечисленным выше сложного контента при одновременном повышении качества за счёт сочетания следующих шаблонов:

- частое применение изменений (сборка)
- упрощение и ускорение сборки
- эащищённая система управления версиями
- уровни сборки и тестирования (частная, групповая)
- песочница(песочницы) для тестирования
- скрипты ускоренного тестирования
- запуск всех тестов на общем компьютере
- оперативное адресное информирование разработчиков о изменениях в коде и об ошибках исполнения их кода в системе
- скриптовое конфигурирование
- упрощение возврата к старой версии

Эффект непрерывной интеграции определяется качеством организации тестирования и связи разработчиков с пользователями.

Цикл "ошибка замечена пользователем - исправлена - проверена" может быть сокращен до нескольких минут.

Обещая снизить цену каждой ошибки (будь то ошибка в коде или в действиях администратора), основанные на ретроспективной СУБД системы могли бы стать намного более эффективны и дружелюбны к разработчикам и пользователям, чем строящиеся на стандартной основе.

Может быть реализован и прозрачный автоматический учёт авторских вкладов участников в код, данные и тестирование в каждом контексте данных.

Подход может рассматриваться и в качестве основы для системы неограниченно возрастающей сложности, долговременно устойчиво развивающейся (“perpetual evolution”) на фоне технических, политических и юридических преображений. Новые исходные требования к такой системе лаконично сформулированы институтом SEI (Software Engineering Institute) в [4,5] в виде ключевых особенностей широкомасштабируемых социотехнических систем будущего (ULS):

- Децентрализация;
- Противоречивые по своей сути, непознаваемые, и разнообразные требования к системе;
- Непрерывное развитие и развёртывание;
- Разнородные, рассогласованные и изменяющиеся элементы;
- Стирание границы между людьми и системой;
- Устойчивость к ошибкам в исходных данных и при их обработке;
- Соревнование за системные ресурсы;
- Наличие организаций и участников, ответственных за установление политик;
- Наличие организаций и участников, ответственных за производство самой системы;
- Локальные и глобальные показатели здоровья, которые будут подключать необходимые изменения в политиках и в поведении элемента и системы;
- Дизайн и эволюция производственных отношений;

- Оркестровка и управление;
- Наблюдение и оценка.

Сформулировав перечисленные требования, рабочая группа SEI дала им недвусмысленную оценку:

“These characteristics undermine the assumptions we make in most current technical, management, and acquisition approaches.”

“Today’s approaches are based on perspectives that fundamentally do not cope with the new characteristics arising from ultra-large scale. The mentality of looking backward doesn’t scale.”

Такая оценка несомненно ориентирует на пересмотр стандартных подходов.

Напрашивается гипотеза о том, что создание и развитие контекстно-автономных систем [7] указывает значительно более простой, просматриваемый и экономный путь к ULS, чем предложенное SEI развитие сервер-ориентированной архитектуры.

Подтвердить или опровергнуть эту гипотезу могут только прикладные исследования.

3. Принципиальные особенности подхода

3.1 Принцип персика

Система, удовлетворяющая требованиям децентрализации, непрерывного развития и развёртывания, содержащая разнородные, рассогласованные и изменяющиеся элементы и устойчивая к ошибкам в исходных данных и ошибках исполнения может, подобно персику, состоять из ядрышка, косточки и мякоти.

Ядрышко — это защищённые данные системы, включая те, которые уже не актуальны, но ещё могут потребоваться для диагностики и исправления ошибок. Эти данные должны храниться с обеспечивающим надёжную сохранность дублированием распределённо, надёжнее в разных ЦОДах, возможно под управлением различных операционных систем, синхронизироваться и распределяться в вновь создаваемые хранилища. Ядрышко неограниченно количественно и качественно растёт в ходе жизнедеятельности системы, включая в себя все данные и логику функционирования приложений. Оно надёжно защищено от противоречий, рассогласованностей и ошибок жёстко определённой косточкой семантикой низкого уровня.

Косточка — это особо выверенный и жёстко защищённый исполняемый код, обеспечивающий защищённый доступ к данным и гарантирующий их целостность на фоне штатных сбоев и поломок оборудования.

Он использует прозрачно специфицированные протоколы и алгоритмах распределённого хранения и синхронизации изменений, выявления и удаления заведомо ненужной информации, регламенты аварийно-восстановительных работ.

Этот код минимален и предельно прост, тщательно выверен и отлажен, устойчив к любым ошибкам в остальных частях и не требует коррекции при любых изменениях логики приложений.

Мякоть — это прикладная логика системы, адекватно отражающая потребности пользователей. Как и эти потребности, она может быть локальна, непоследовательна, временами противоречива, по разному представляться разным группам и пользователям. Она вечно неограниченно эволюционирует, местами иногда возвращаясь к прежним состояниям. Эта логика выполняется в фоновом защищённом режиме и любые ошибки её задания или исполнения имеют жёстко ограниченные косточкой во времени и информационном пространстве последствия.

3.2 Ретроспективная основа

Основным средством защиты системы от ошибок в прикладных обработках становится опора на эффективный доступ к прошлой информации системы. Полная история изменений информации — это и есть ядрышко, нуждающееся в тщательной надёжной защите.

Любой запрос на чтение новой информации мгновенно возвращает установившееся в системе прошлое состояние и остается гарантированно воспроизводимым с тем же результатом.

Если запрос к системе адресован к медленно обрабатываемым изменчивым данным, то система не заставит ждать окончания обработки, а немедленно выдаст последнюю готовую версию с индикацией момента времени актуальности и ставит запрос в очередь обработки.

Такой пока непривычный интерфейс позволит значимо ускорить работу с системой, задержки в работе которой неустранимы по причине больших расстояний или сложных алгоритмов.

Ремонт бизнес-логики воспринимается пользователем как замедление обработки новых данных. Сопровождающий код программист оперативно извещается о поломке, если он быстро её устраняет, то пользователи не замечают дефекта.

Требование устойчивости к ошибкам будет сохранять актуальность и по чисто технической причине [8], [9]: уменьшение размеров транзисторов и их энергопотребления при повышении быстродействия приводит к усилению эффектов квантовой механики, неотвратимо повышающих вероятность ошибки в конкретной ячейке памяти. Разумеется, ошибки в основном компенсируются использованием дублирования и особого кода, исправляющего ошибки, но требующего дополнительных ресурсов. Предлагаемая архитектура предъявляет высокие требования к хранению информации и исполнению

регламентных процедур косточки. Обработки данных приложений в ряде случаев могут проводиться быстрее и с меньшей надёжностью, если выходные данные в контексте должны подвергаются независимому контролю и при необходимости перевычислению.

3.2 Контекстная автономность

Децентрализация не означает анархии, она означает иерархию ответственности. По принципу единоначалия, один человек отвечает за систему в целом, но поскольку большая система состоит из подсистем, то он делегирует управление и ответственность за некоторые или все подсистемы своим подчинённым, которые могут продолжить этот процесс. Таким образом, всё информационное пространство системы предстаёт иерархией автономно управляемых подпространств-контекстов данных.

Смысл существования каждого такого контекста в специфическом информационном обслуживании, то есть в обеспечении клиентов качественно обработанной информацией. Хозяин контекста имеет полный доступ к информации своего автономного контекста, устанавливает внутри него политики и правила доступа, выбирает доступные способы обработки, разграничивает доступ и регулирует оркестровку и персонификацию, выбирает и использует средства мониторинга и оценки, делегирует эти функции подчинённым (или роботам), при необходимости отделяя для них дочерние автономные контексты, устанавливает роли, политики и правила вычисления относительных приоритетов очерёдности обработки данных.

Любая часть выходной информация контекста может быть объявлена доступной для любого множества контекстов. Хозяину всегда доступна информация, сколько пользователей и насколько часто используют те или иные выходные интерфейсы напрямую или через какие контексты.

Косточка регламентирует, где в ядрышке должна находиться информация о контекстах, делегировании, ролях, приоритетах и настройках ресурсов, как собирается статистика использования и организуется выполнение обработки информации и обеспечивает эти работы в точном соответствии с регламентом. Мониторинг ресурсов настраивается в рамках каждого автономного контекста с использованием явных и неявных оценок качества кластерами потребителей.

Контекстно-автономный подход резко усиливает составляющие непрерывной интеграции, в корне изменяя привычные представления и ориентиры в разработке.

3.4 Жизненный цикл данных

Хранение истории исходных данных, промежуточных и окончательных результатов обработки требует затрат, существенно зависящих от способа хранения и поэтому трудно поддающихся оценке.

Однако при интенсивном обновлении данных на практике неизбежно возникает проблема недостатка пространства для хранения данных. Хранение промежуточных и окончательных результатов обработки типовых запросов усугубляет ситуацию, вынуждая избавляться от старых никому не нужных данных.

Чтобы удаление данных не нарушило требуемую целостность системы, предполагалось выделять в далёком прошлом короткие промежутки времени, так называемые «белые пятна истории», связанные с интенсивным изменением данных, и удалять версии данных, не актуальные вне этих промежутков. Поскольку выделение таких пятен при больших объёмах данных является непростой задачей, то предлагалось решать её в рамках отделльных автономных контекстов, что могло нарушить согласование данных между разными контекстами.

Сейчас найдено простое решение, позволяющее чистить ненужную часть истории без перерасхода ресурсов и угрозы рассогласования. Оно состоит в увеличении шага дискретизации для очень старых данных. Например, если шаг дискретизации для данных более чем десятилетней давности переустанавливается в одни сутки, то из всех изменений любого данного в течение таких давно прошедших суток оставляется последнее, актуальное на момент полуночи. Все изменения, актуальные на границе каких-либо суток, остаются в доступе, а все очень старые изменения, не дожившие до ближайшей полуночи, теряются безвозвратно.

3. План разработки

При стандартном подходе к разработке бизнес-логика выявляется в предпроектном обследовании. Но логику сложной творческой деятельности практически невозможно заранее чётко выяснить и формализовать. В идеале она вырастает в ходе совместной деятельности разработчиков и субъектов самого бизнес-процесса и гибко меняется при необходимости. Поэтому бизнес-логика приложения не закладывается в основу ИС, использующей ретроспективную СУБД, а выращивается в процессе использования.

Таким образом, разработка очередной системы описываемого класса должна быстро и просто базироваться на подходящем прототипе, в котором постепенно будет корректироваться и наращиваться требуемая бизнес-логика.

Создание «косточки» для первого такого прототипа, способного эффективно координировать действия разработчиков и пользователей, является сложной научно-технической проблемой. Используемые для создания стандартных систем инструменты,

заготовки и архитектурные решения практически практически полностью непригодны для достижения поставленной цели. Требуется в комплексе решить ряд непростых тесно взаимосвязанных специфических задач, не имеющих отношения к стандартным технологиям.

3.1 Разработка распределённой масштабируемой B+tree СУБД

Особенность предъявляемых требований к B+tree СУБД в том, что от неё не требуется *модифицировать записи или удалять недавно созданные.* Для рассматриваемого класса СУБД каждая запись состоит из ключа и значения. Ситуация, в которой требуется записать старый ключ с новым значением может возникнуть только при редкой ошибке и какое именно значение окажется у ключа в этом случае не важно. Поэтому порядок, в котором производятся записи, не существенен и это снижает конкурентность записи, существенно облегчая задачу синхронизации данных. Синхронизация достаточно просто сводится к защищённому от потерь обмену списками новых записей.

Основная проблема здесь состоит в необходимости предоставления беспрерывного доступа на чтение в то время как база пополняется свежими данными. Такая возможность была документирована в BerkeleyDB4.2, но в более поздних версиях разработчики отказались от поддержки этой возможности. Сейчас она документирована только в TokyoCabinet, но тест качества поддержки этой функциональности пока не проведён. При этом база TokyoCabinet периодически нуждается в дефрагментации, во время которой она недоступна на чтение.

В [] предложена идея каскадного использования B+tree СУБД, позволяющая за счёт одновременного использования и подготовки нескольких баз вместо одной получить требуемую функциональность на основе любых СУБД, поддерживающих требуемый *поиск ближайшего ключа* к произвольной строке запроса (имеется в виду близость в лексикографически упорядоченном списке ключей, пополненном строкой запроса). Идея состоит в формировании маленьких баз, содержащих дополнения за долю секунды и слияния их в фоновом режиме в более крупные с дефрагментацией и оптимизацией. Интерфейс СУБД при обработке запроса проверяет, не устарел ли список баз в памяти системного процесса или нити исполнения и при необходимости обновляет этот список. Устаревшие базы удаляются в фоновом режиме. Завершение отладки и тестирования позволит говорить о выполнении этого пункта плана.

Добавление к ключу каждого записываемого данного времени нулевого байта, строки, лексикографически упорядоченно идентифицирующей момент актуальности, возможно, ещё одного нулевого байта и строки, идентифицирующей момент окончания обработки, превращает B+tree СУБД, удовлетворяющую выделенным курсивом требованиям, в ретроспективную.

3.2 Создание "живого словаря"

«:Живой словарь» — это простая ретроспективная СУБД, представляющаяся основой и главным инструментом для создания первой контекстно-автономной системы. Дополнительно к описанной в **3.1** функциональности она предоставляет:

1. Индикацию наличия изменений в данных ветки за любой период.
2. Представление редактируемых структур, включающих хеши, и массивы данных, в качестве веток.
3. Навигацию по дереву данных, аналогичную панели файлового эксплорера.
4. Ретроспективную индексацию данных.

Первое более эффективно реализуется не циклом, а использованием служебного ключа, отвечающего за список изменённых в ветке ключей. Такой ключ может как создаваться для любой ветки, так и удаляться с переносом информации в ближайшую содержащую ветку с таким ключом.

Второе означает описание служебных структур, используемых для экономного хранения редактировавшихся структур (перестановка, удаление или вставка элементов массивов и пр.)

Третье означает возможность понять, в каком месте дерева данных мы находимся и что в нём лежит. Проще всего было бы использовать в качестве ключа полный путь к узлу, но с длинными идентификаторами трудно работать. Поэтому полное имя каждого узла или ветки для низкоуровневой навигации должно быть сохранено в системном словаре. Оно не обязательно должно быть уникальным: полный путь к узлу может разбиваться на части, имеющие уникальные имена.

Возникает задача исключить появление ложных ассоциаций и присвоение непредусмотренных имён, естественный путь к решению которой – согласованность алгоритма выбора разбиения пути на именованные части с алгоритмом выбора идентификатора для новой ветки.

Для добавления новых имён и поиска нужного узла необходима индексация системного словаря в специальной части основной базы. Сложность в ретроспективности, автоматически обновляемый словарь должен корректно отвечать на запрос по истории.

«Живой словарь» находится в процессе разработки. Отдельные функциональности реализованы на макетах, но требуется их одновременная реализация.

3.3 Разработка подсистемы неограниченно масштабируемых многослойных структур данных.

Базируясь на "живом словаре", эта подсистема должна обладать всеми качествами "живого словаря" и при этом обеспечивать

1. Аналог символических ссылок файловой системы с функциональностью команды "append" из msdos с логикой наследования (если записи с заданным ключом нет, но укороченный ключ является ссылкой, то ищется запись по ссылке и т.д.);
2. Экономное (без дублирования) сосуществование нескольких версий хранимой информации (вариантов упорядочения массива, веток с по-разному привитыми подветками и т. д.)

3.3 Разработка подсистемы контекстно-автономного управления изменениями:

- выделение автономных контекстов;
- регистрация изменений в контексте и чтений из него, фиксации установившегося состояния;
- организация очереди применения изменений с приоритетной ориентацией на ожидаемые результаты.

3.4 Разработка фреймворка персонально оптимизируемых пользовательских интерфейсов, ориентированных на работу с устоявшимися состояниями системы, её историей и её гладкую доработку.

3.5 Рефакторинг кода, расширение масштабов использования и сфер приложений в духе принципа персика.

4 Заключение

Функциональность косточки должна быть предельно простой, но достаточной для обеспечения защищённости и целостности ядрышка. Поэтому модель данных ядрышка должна быть простейшей из обещающих требуемую функциональность.

Литература

[1] by Paul M. Duvall. Continuous Integration: Improving Software Quality and Reducing Risk, Addison-Wesley, 2007. http://www.amazon.com/gp/product0321336380/?tag=integratecom-20

[2] Знаменский С. В. Хорошо масштабируемое автономное администрирование доступа. Труды Международной конференции "Программные системы: теория и приложения", Переславль-Залесский, октябрь 2006, Наука,-Физматлит, М. Т. 1, с. 155-169. ---.

[3] http://skif.pereslavl.ru/psi-info/psi/psi-publications/e-book-2006/index.html

[4] С. В. Знаменский. Гибкая основа информационной системы для обучения. //Труды XII-й Всероссийской научной конференции «Электронные библиотеки: перспективные методы и технологии, электронные коллекции» RCDL-2010. Казань: Казанский университет. 2010, с. 451-460.

[5] Linda Northrop.The Impact of Scale: Carnegie Mellon University. December 17, 2009 http://www.sei.cmu.edu/library/assets/20091217webinar.pdf

[6] Linda Northrop. Ultra-Large-Scale Systems. The Software Challenge of the Future. June 2006. Pittsburgh, PA 15213-3890 http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf

[7]

[8] Mart Roost, Karin Rava, and Tarmo Veskioja. 2007. Supporting self-development in service oriented information systems. In Proceedings of the 7th Conference on 7th WSEAS International Conference on Applied Informatics and Communications - Volume 7 (AIC'07), Minh Hung Le, Metin Demiralp, Valeri Mladenov, and Zoran Bojkovic (Eds.), Vol. 7. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 52-57.

[9] J.C. Laprie, Dependability: Its Attributes, Impairments and Means. In *Predictably Dependable Computing Systems*, B. Randell, J.C. Laprie, H. Kopetz, and B. Littlewood (eds.), pp. 1-28, Springer-Verlag, 1995.

[10] B. Bloom. Space/time Trade-Offs in Hash Coding with Allowable Errors. In *Communications of ACM*, volume 13(7), pages 422-426, 1970.

.

Factographic base for the complex collaborative information resources reorganization

S.V. Znamenskij

Some Information Resources, especially those addressed the complex issues are known to be continuously improved with a broad range of individuals and organizations.
Such refinement is often needed in sudden corrections of the structure of the resource, parallel development and comparative testing of users of multiple versions of its components, removal of excess and restore previously available data.

We discuss the idea of building a representation of data in such a system on immutable records addressable given date, time and authorship of their creation and providing accelerated multiplayer improvements due to (1) immediate access to the same state information and the essential characteristics of change;
(2) localization errors of execution in time and information space;
(3) the reorganization of a pluralistic view and update data and multi-level testing.

§ Работа поддержана грантом РФФИ № 09-07-00407.