An Oracle White Paper
August 2010

Building Highly Scalable Web Applications with XStream

Disclaimer

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Introduction

Web applications with a very large number of users place an enormous load on the back-end databases for queries as well as transaction processing. As the underlying workload grows, web companies often prefer to partition their data across many databases. Partitioning the data can provide greater scalability but it comes at the expense of consistency and it introduces complexity. Businesses often employ various types of partitioning, such as vertical partitioning (i.e. across functional units) or horizontal partitioning (sharding). However, these types of partitioning strategies must be done carefully to meet overall consistency objectives. Efficient maintenance of the partitions is another important consideration.

This paper describes how XStream's powerful change capture and apply mechanisms can make the implementation of vertical and horizontal partitioning much simpler, faster, and more robust. XStream Out is an API that provides a high-performance streaming interface to capture Oracle database changes with minimal overhead. XStream In is the corresponding streaming interface to deliver a heterogeneous DML workload to Oracle. XStream In applies the DML workload efficiently with parallel processes, and it can apply the workload with optional customizations. The XStream In API provides seamless, position-based restart and recovery, plus a number of filtering and transformation options.

Although XStream is not an end-to-end replication solution, it decouples the capture and apply components that process database changes. Therefore, businesses can insert application logic to build customized solutions.

XStream Overview

Figures 1a and 1b provide a high-level overview of XStream Out and In respectively. Each API has a streaming interface that allows a high throughput of logical change records (LCRs – a generalization of single-row DMLs and DDLs) even over wide area networks. The LCRs include a monotonically increasing position that provides infrequent acknowledgments. The position is also used for recovery and restart of the client application.

An XStream outbound server reads relevant database changes (based on rules) from the redo logs and assembles them into transactions before streaming them to the client application. It has minimal overhead on the source workload, and it can start automatically whenever the client connects to it.

Numerous use cases apply to the XStream Out database event stream. This document describes one use case for decoupling databases. Other uses include complex event processing and cache invalidation.

An XStream inbound server reads a stream of LCRs, grouped into transactions, and applies them in parallel. It also computes and respects dependencies between transactions and enables a configurable degree of parallelism. XStream In can apply the workload with various customizations. For example, a statement DML handler can specify that an arbitrary SQL statement be executed with bind variables using values from the original LCR. Automatic conflict resolution can handle simple conflicts, and more powerful conflict resolution can be written in PL/SQL. When no customizations are specified, XStream In applies each LCR as a database change by default. In some cases, XStream In applies the changes using low-level database functionality, which provides dramatically better performance than applying the changes through SQL.

One way to use XStream In is as a generalized alternative to the batching interfaces provided by Oracle Call Interface (OCI). Unlike batching in OCI, XStream In allows the client to stream a workload of mixed DML types (inserts, updates, and deletes) on multiple tables. Further, the built-in, automatic parallelism allows clients to exploit database concurrency without writing multi-threaded code.

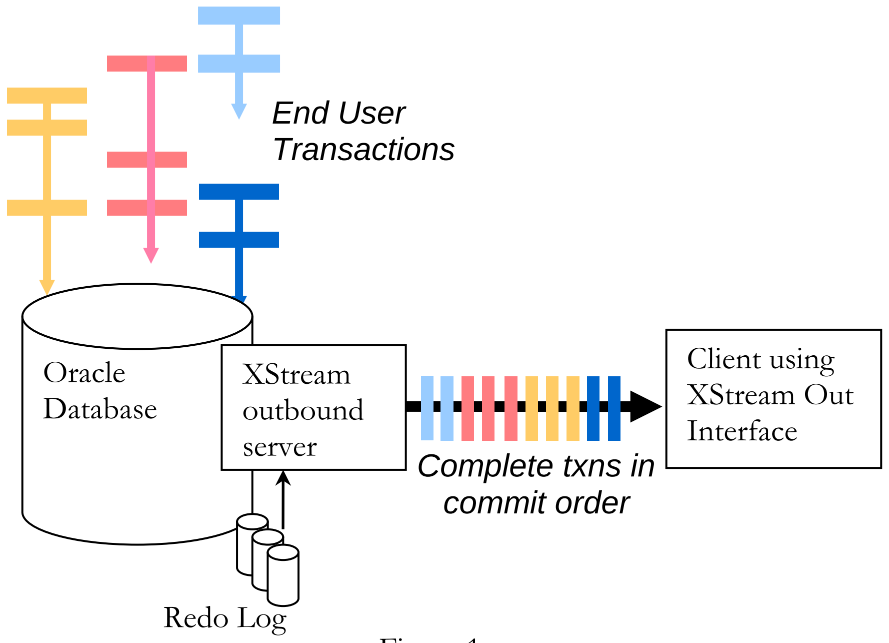

Figure 1a

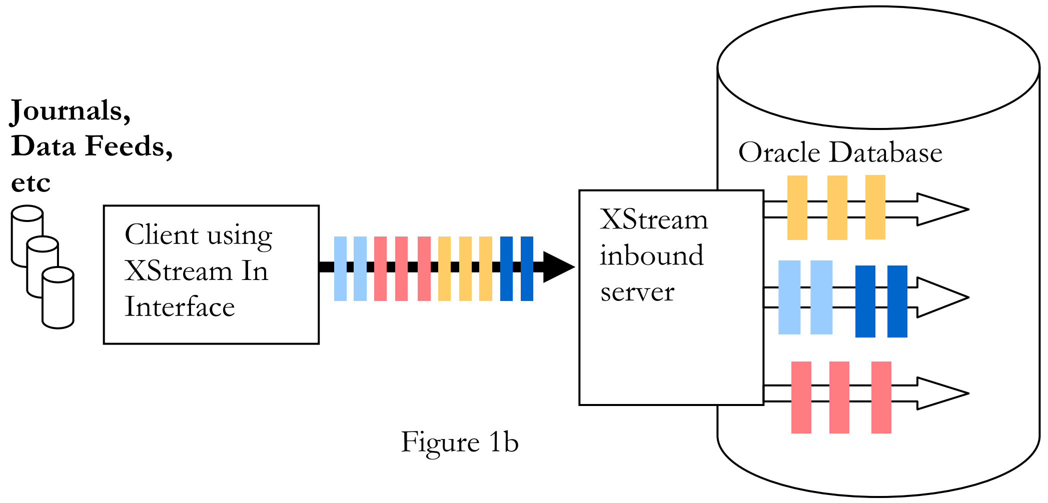

Figure 1b

A Simple Web Application

A typical web application involves queries on mutable data as well as user transactions that must be recorded and processed. We model this interaction in a simple web storefront application. We assume that there is an `item` table that stores all the products marketed by a business. Users query this table through a web interface and place orders for products. The orders are stored in a shopping cart maintained by the web application. Finally, when a user confirms an order, the application processes the shopping cart's contents. The order is inserted into the `order` and `order_line` tables, and the customer's balance in the `customer` table is updated to reflect the money spent. Further, the `item` table is updated to reflect the reduced quantity of the products purchased. The description of the schema is shown in Figure 2a, and the corresponding user transaction is shown in Figure 2b. Note that, independent of the user transactions, the business can update the descriptions and quantities of the items and insert more items.

The transaction shown in Figure 2b dutifully maintains the consistency between the items sold and items on hand, as well the amount spent by the customer and the customer's remaining balance. It might appear that this is indeed the correct way to design this application. However, this transaction introduces some critical bottlenecks. Every customer order requires locking the row corresponding to each purchased product. If a large number of customers try to buy the same product concurrently, then the performance of the system degrades.

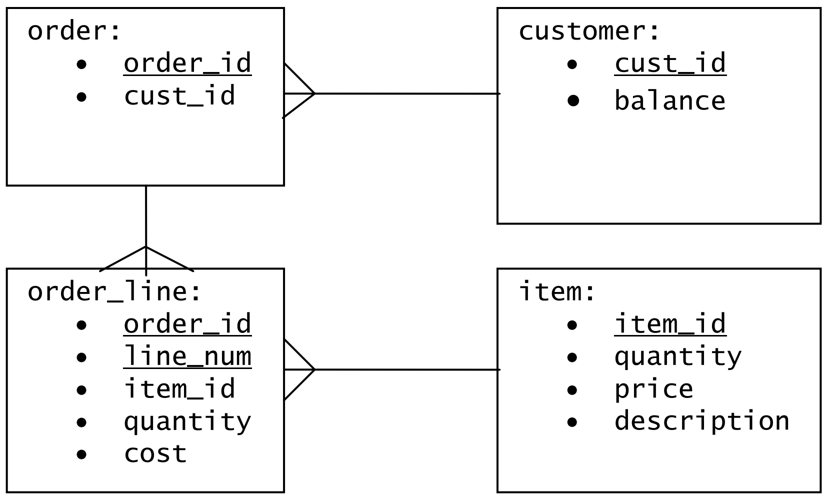

Figure 2a

```
Input: Shopping Cart with cust_id, order_id and a list of item ids and quantities
bill = 0
line_num = 0
for each cart_item_id, cart_item _quantity in cart
    select quantity, price from item where item_id = cart_item_id for update
        if quantity < cart_item_quantity
            raise Error("Sold out")
        else
            update item set quantity = quantity – cart_item_quantity
            where item_id = cart_item_id
    line_num = line_num + 1
    cost = cart_item _quantity * price
    insert into order_line(order_id, line_num, cart_item_id, cart_item_quantity,
                                cost)
    bill = bill + cost

select balance from customer where cust_id = cart_cust_id for update
if balance < bill
    raise Error("Insufficient funds")
else
    update customer set balance = balance – bill

insert into order(order_id, cust_id)
commit
```

Figure 2b

Decoupling Orders from Items and Customers

A natural solution to reduce the contention on the `item` and `customer` tables is to not update them with every order, as shown in Figure 3.

```
Input: Shopping Cart with cust_id, order_id and a list of item_ids and quantities
bill = 0
line_num = 0
for each cart_item_id, cart_item _quantity in cart
   select quantity, price from item where item_id = cart_item_id ~~for update~~
      if quantity < cart_item_quantity
         raise Error("Sold out")
      ~~else~~
         ~~update item set quantity = quantity – cart_item_quantity~~
         ~~where item_id = cart_item_id~~
   line_num = line_num + 1
   cost = cart_item _quantity * price
   insert into order_line(order_id, line_num, cart_item_id, cart_item_quantity,
                          cost)
   bill = bill + cost

select balance from customer where cust_id = cart_cust_id ~~for update~~
if balance < bill
   raise Error("Insufficient funds")
~~else~~
   ~~update customer set balance = balance – bill~~

insert into order(order_id, cust_id)
commit
```

Figure 3

This leaves the task of maintaining the `item` and `customer` tables to a secondary transaction processing system (in the same database). This secondary processing can be implemented easily with XStream. Figure 4a demonstrates a simple application in C or Java in pseudo code. The application connects to an XStream outbound server and receives all the transactions that modify the `order` and `order_line` tables. For each `order_line` insert, the code constructs a corresponding update to the `item` table. Maintaining the `customer` table requires a little more work since the update to the customer balance depends on the total cost in all of the `order_line` inserts. Hence the update to the `customer` table is generated after processing the entire transaction. Finally, the client application sends these updates to an XStream inbound server, which applies them through a statement DML handler (Figure 4b) that converts them into delta updates. The position for the transaction given to the inbound server is copied from the corresponding transaction received from the outbound server. This reuse of a position simplifies recovery. On startup, the application gets its position from the inbound server and then gives this position to the outbound server to resume processing. Thus the application can be stateless and still achieve exactly-once semantics.

Although Figure 4a deals with inserts only, different update and delete statements on the `order_line` and `order` tables can be handled in a similar fashion.

```
Get position from XStream inbound server
Give position to XStream outbound server
For each transaction Tout received from outbound server
  Construct transaction Tin
  bill = 0
  For each record in Tout
    If record is insert order_line(order_id, line_num, item_id, quantity, cost)
      bill = bill + cost
      add update item (:old.item_id=item_id, :new.quantity=quantity) to Tin
    If record is insert into order(order_id, cust_id)
      saved_cust_id = cust_id

  add update customer (:old.cust_id=saved_cust_id, :new.balance=bill) to Tin
  Tin.position = Tout.position
  Give Tin to inbound server
```

Figure 4a

```
Update item set quantity = quantity - :new.quantity where item_id=:old.item_id

Update customer set balance = balance - :new.balance where cust_id=:old.cust_id
```

Figure 4b

Decoupling the tables achieves higher throughput, but this improvement comes at the cost of consistency. The customer balance or an item quantity is now allowed to become negative (although this should still be rare). We assume that the business model handles such uncommon inconsistencies.

Sharding Customer and Item tables

So far we have reduced the amount of work done by the customer facing application and the associated contention but not the total amount of work done in a single database. Also, we have not reduced the query load on the `item` and `customer` tables generated by the web application. A logical next step, therefore, is to partition the `customer` and `item` tables across many databases by static partitioning on the primary key. Further, since customers might want to query their orders as well, the `order` and `order_line` tables are partitioned so that the orders for a given customer are stored in the database that has the corresponding `customer` row. This configuration is shown in Figure 5. (For simplicity, we are partitioning both the `customer` and the `item` table into *n* partitions, but we could have chosen a different number for each.)

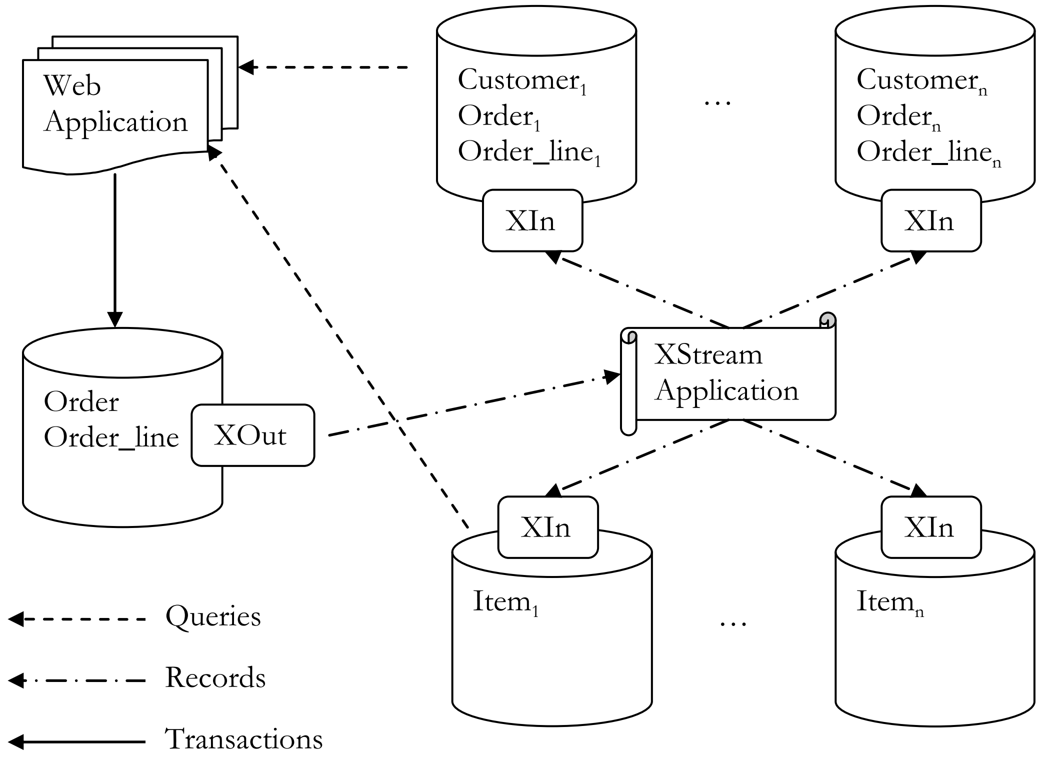

Figure 5

The new XStream application in Figure 6 is similar to the one in Figure 4, except that it must split each incoming transaction into multiple transactions, one for each of the

```
Get position from each XStream inbound server
Give minimum position to XStream outbound server
For each transaction Tout received from XStream Out
  Construct transactions Tin-Cust, and $Tin-Item_1$, …, $Tin-Item_n$
  bill = 0
  For each record in Tout
     If record is insert order_line(order_id, line_num, item_id, quantity, cost)
        bill = bill + cost
        add record to Tin-Cust
        add update item (:old.item_id=item_id, :new.quantity=quantity)  to $Tin-Item_{hash(item\_id)}$
     If record is insert into order(order_id, cust_id)
        saved_cust_id = cust_id
        Add record to Tin-Cust

  add update customer (:old.cust_id=saved_cust_id, :new.balance=bill) to Tin-Cust

  {Tin-Cust, $Tin-Item_1$,..., $Tin-Item_n$}.position = Tout.position
  Deliver Tin-Cust to the inbound server in customer database-hash(saved_cust_id)
  For each non-empty $Tin-Item_1$, …, $Tin-Item_n$
     Give the transaction to inbound server in the corresponding Item database
```

Figure 6

databases that contains the `item` or `customer` row. A *hash* function, which returns a number from 1 to n, determines the partitioning. There is a slight subtlety when routing the `order_line` table. Unlike the other tables, this one does not store the partitioning key, which in this case is `cust_id`, and hence it cannot be routed by itself. However, it is one of the strengths of the XStream approach that secondary transaction processing can examine the contents of the entire transaction. In this example, the `cust_id` column is extracted from the insert to the `order` table, and the transaction is routed appropriately. For the `item` and the `customer` tables, the new application uses the statement DML handler from Figure 4b. The position for restarting the XStream outbound server is now a minimum of all the XStream inbound server positions. This means that some of the inbound servers may get duplicates on restart. However, the inbound servers automatically suppress the duplicates.

Figure 5 shows a single database that receives transactions on the `order` and `order_line` tables from the web application. If this database becomes a bottleneck, then multiple databases can be configured. The web application servers can distribute their load over these databases. Each "front-end" database needs a dedicated XStream outbound server, an XStream client application, and XStream inbound servers at each "back-end" database that stores the customer and item tables. The various inbound servers in the "back-end" databases do not conflict with each other because their workload consists of inserts and delta-updates.

Technical Considerations

We could have used plain SQL statements to apply the changes to the various tables rather than using XStream In. However, if we had used SQL, then we would have to configure various optimizations to ensure good performance. For example, we would need to batch the SQL statements to minimize round trips and write a multi-threaded client to maximize the concurrency. XStream In automatically handles both of these concerns for us. XStream In also maintains the position of the incoming stream that it has consumed, and this position maintenance allows our client to be stateless.

The decoupling of databases that we have proposed in this paper might appear similar to traditional replication. However, there is an important difference. Unlike typical row-level replication that allows filtering and transformations based on single row changes, XStream In allows the client to inspect the contents of the entire transaction before determining how to process it. In our example, we could not have determined which database to send the `order_line` row without examining the complete transaction. Similarly, we needed to examine the complete transaction to determine the amount to bill the customer. In more complicated examples, the user transactions can easily write to auxiliary tables if necessary to help the XStream client application determine the appropriate action.

Sample code for the XStream application in Figure 6 and instructions for configuring the databases as shown in Figure 5 are available in the demo/xstream/scalewp directory of Oracle Database 11.2.0.2.

Conclusion

We have demonstrated that with the help of XStream a single database can be easily decoupled into multiple databases to achieve boundless scalability.

Oracle is committed to developing practices and products that help protect the environment

Building Highly Scalable Web Applications with XStream
August 2010
Author: Nimar S. Arora
Contributing Authors: Lik Wong, Patricia McElroy, James Stamos, Vinoth Chandar, Haobo Xu, Byron Wang, and Randy Urbano.

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Copyright © 2010, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd. 0410

SOFTWARE. HARDWARE. COMPLETE.