Resource Efficient Parallel VLDB with Customizable Degree of Redundancy

A Dissertation

Submitted to

The Temple University Graduate Board

in Partial Fulfillment

of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

By

Fanfan Xiong

May, 2009
ABSTRACT

Databases have become critical part of human lives. As their sizes multiply annually, their importance grows exponentially. Ironically, although database servers hold the vast majority critical data, they are the hardest to scale up and cost prohibitive to protect.

This thesis focuses on the practical use of very large scale relational databases. It leverages two recent breakthroughs in parallel and distributed computing: a) synchronous transaction replication technologies by Justin Y. Shi and Suntain Song; and b) Stateless Parallel Processing principle pioneered by Justin Y. Shi. These breakthroughs have enabled scalable performance and reliability of database service using multiple redundant shared-nothing database servers. In this thesis, we present a methodology with customizable degree of redundancy to address practical very large scale database applications problems.

This dissertation focuses on a resource-efficient customizable VLDB parallel replication system. It aims to deliver the following:

a) A customizable VLDB parallel partition and replication strategy that will allow automatic dataset partition, data distribution and seamless (non-stop) recovery.

b) An application programming interface that will automatically translate user queries using virtual global schema into parallel queries targeting multiple physically partitioned schemas.
The central theme is to optimize overall efficiency of a VLDB application with variable degrees of redundancy.

The prototype VLDB implementation relies heavily on a prior work named DBx by Dr. Shi and Suntain Song. DBx is a general-purpose VLDB cluster runtime substrate capable of delivering high performance and high reliability at the same time using multiple shared-nothing database servers. It was designed for transparent non-intrusive deployments. Due to the critical nature of the subject matter, we believe our findings will be useful references to researchers in the database community in the large.

This dissertation focuses on a new data partitioning technology to allow a resource-efficient implementation and its supporting application programming interface. The prototype system supports Microsoft SQL Servers databases. Computational experiments are conducted using industry-standard benchmarks.
ACKNOWLEDGMENTS

I wish to express my sincere appreciation to Dr. Justin Shi, Dr. Vasilis Megalooikonomou, Dr. Museke Sendaula and Dr. Alexander Yates for their guide and instruction during the research and the preparation of this manuscript. In addition, special thanks to Ms. Jacalyn Harriz, whose familiarity of the needs was helpful during my early study phase.

Finally, I thank my wife Lei Wang and other family members – Xianglu Xiong, Jucui Ma, Pupu Xiong, Jing Wang, Duoduo Xiong, Sydney Y. Xiong – and all my Landlord’s family members – Mr. Sidney A. Friedman (1935-2003), Mrs. Sue H. Friedman, Lori Friedman, Wendi Friedman, Daisy Friedman and Petti-Cake Friedman – for their love and support during my study. Their love, patience, and encouragement made this dissertation possible. I affectionately dedicate this dissertation to them.

Cherry Hill, New Jersey

Fanfan Xiong

May 2009
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>x</td>
</tr>
<tr>
<td>Chapter 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Prior Work</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Statement of Research Problem</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Envisioned Contributions</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Thesis Organization</td>
<td>5</td>
</tr>
<tr>
<td>Chapter 2 BACKGROUND</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Transaction Load Balancing</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Transaction Replication Technologies</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Serial Replication Technologies</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Parallel Replication Technology</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Data Partitioning</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1 Horizontal Partitioning and Vertical Partitioning</td>
<td>12</td>
</tr>
<tr>
<td>2.3.2 Internal Partitioning and External Partitioning</td>
<td>15</td>
</tr>
<tr>
<td>2.3.3 Partitioned View and Partitioned Table</td>
<td>17</td>
</tr>
<tr>
<td>2.3.4 Partitioning Criteria</td>
<td>19</td>
</tr>
<tr>
<td>2.4 Parallel Replicated Partitioned DBMS: RAIDB</td>
<td>20</td>
</tr>
<tr>
<td>2.5 Markup Language for Transaction Replication Technologies</td>
<td>21</td>
</tr>
<tr>
<td>2.5.1 RML Overview</td>
<td>21</td>
</tr>
<tr>
<td>2.5.2 RML Parser</td>
<td>23</td>
</tr>
<tr>
<td>2.5.3 New Extension on Partitioning - PRML for RAIDB System</td>
<td>24</td>
</tr>
<tr>
<td>2.6 Summary</td>
<td>25</td>
</tr>
<tr>
<td>Chapter 3 DESIGN OF RAIDB WITH VARIABLE DEGREE OF REDUNDANCY</td>
<td>27</td>
</tr>
<tr>
<td>3.1 Challenges and Expected Output</td>
<td>29</td>
</tr>
<tr>
<td>3.2 TPC-E Example Database</td>
<td>32</td>
</tr>
<tr>
<td>3.3 Schema Design of VLDB with Variable Degree of Redundancy</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1 Outline of VLDB with variable redundancy</td>
<td>36</td>
</tr>
</tbody>
</table>
3.3.2 Data Partitioning for RAIDB system using PSR Middleware gateway ........................................ 40

3.3.3 Functional Horizontal Partitioning ............................................................................................ 46

3.4 Data structure and API for PSR system with Customizable partitioning ......................................... 49

3.4.1 Data Structure ............................................................................................................................ 49

3.4.2 API functionalities ....................................................................................................................... 52

3.5 Summary ........................................................................................................................................ 53

Chapter 4 PRML FOR DATABASE WITH VARIABLE REDUNDANCY .............................................. 54

4.1 PRML Tag for Functional Horizontal Partitioning ......................................................................... 55

4.1.1 Single-Partition Tags ................................................................................................................ 56

4.1.2 Cross Partition Scan Tag (READFULL tag) ............................................................................ 59

4.1.3 Replication control tag (WRITETOALL tag) ........................................................................... 60

4.2 An Example of PRML ..................................................................................................................... 62

4.3 Summary ....................................................................................................................................... 63

Chapter 5 PARSEING PRML TAGS ................................................................................................. 64

5.1 The PRML Tag Priority and PRML Mask ..................................................................................... 65

5.2 The PRML Parser and Flow Control ............................................................................................ 66

5.3 Lock Isolation Control by PRML statement ................................................................................ 67

5.4 Summary ....................................................................................................................................... 70

Chapter 6 TRANSACTION CONTROL IN PSR SYSTEM .................................................................. 72

6.1 Transaction Consistency Control .................................................................................................. 72

6.2 Transaction Resubmission during Gateway Failover .................................................................... 73

6.2.1 Transaction Resubmission without Connection Pool ............................................................... 75

6.2.2 Transaction Resubmission with Connection Pool .................................................................... 76

6.3 Use Non-Deterministic Functions in RAIDB System .................................................................. 77

6.4 Summary ....................................................................................................................................... 80

Chapter 7 PERFORMANCE MODELING AND COMPUTATION RESULTS .................................... 81

7.1 Throughput Speedup of RAIDB system ....................................................................................... 81

7.2 Timing Models for RAIDB System ............................................................................................... 86

7.2.1 Timing Model for Insert, Update and Delete Operations ......................................................... 86

7.2.2 Timing Model for Select Operations ....................................................................................... 87

7.3 Logical Partition and Virtual Partition in RAIDB System ............................................................. 89

7.4 Summary ....................................................................................................................................... 92

Chapter 8 EXPERIMENTAL RESULTS .............................................................................................. 93
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>RAIDB Conceptual Design</td>
<td>4</td>
</tr>
<tr>
<td>2-1</td>
<td>PSR on Traditional Horizontal Partitioning</td>
<td>14</td>
</tr>
<tr>
<td>2-2</td>
<td>Traditional Horizontal Partitioning</td>
<td>16</td>
</tr>
<tr>
<td>2-3</td>
<td>Example of Partitioned View</td>
<td>17</td>
</tr>
<tr>
<td>2-4</td>
<td>Example of RML Statements</td>
<td>23</td>
</tr>
<tr>
<td>2-5</td>
<td>RML Flow Control</td>
<td>24</td>
</tr>
<tr>
<td>3-1</td>
<td>Thrashing in RDMS</td>
<td>29</td>
</tr>
<tr>
<td>3-2</td>
<td>Expected Performance Gain of Shared-Nothing Cluster</td>
<td>31</td>
</tr>
<tr>
<td>3-3</td>
<td>E-R Diagram of TPC-E Benchmark Database</td>
<td>32</td>
</tr>
<tr>
<td>3-4</td>
<td>Business Model of Custom Call Center</td>
<td>34</td>
</tr>
<tr>
<td>3-5</td>
<td>Partition using Customizable Degree of Redundancy</td>
<td>36</td>
</tr>
<tr>
<td>3-6</td>
<td>Procedure of Functional Horizontal Partitioning Implementation</td>
<td>37</td>
</tr>
<tr>
<td>3-7</td>
<td>Partition Data whining Single Server</td>
<td>38</td>
</tr>
<tr>
<td>3-8</td>
<td>Partitioned Data Migrate</td>
<td>38</td>
</tr>
<tr>
<td>3-9</td>
<td>Regular Publisher/Subscriber</td>
<td>39</td>
</tr>
<tr>
<td>3-10</td>
<td>Categories of TPC-E Tables</td>
<td>42</td>
</tr>
<tr>
<td>3-11</td>
<td>Dependency Tree of TPC-E Database (1)</td>
<td>47</td>
</tr>
<tr>
<td>3-12</td>
<td>Dependency Tree of TPC-E Database (2)</td>
<td>48</td>
</tr>
<tr>
<td>3-13</td>
<td>Example of FHP Configuration (1)</td>
<td>51</td>
</tr>
<tr>
<td>3-14</td>
<td>Example of FHP Configuration (2)</td>
<td>51</td>
</tr>
<tr>
<td>3-15</td>
<td>Example of FHP Configuration (3)</td>
<td>52</td>
</tr>
<tr>
<td>4-1</td>
<td>Example of READSLIM Tag</td>
<td>57</td>
</tr>
<tr>
<td>4-2</td>
<td>Example of WRITESLIM Tag</td>
<td>58</td>
</tr>
<tr>
<td>4-3</td>
<td>Example of Nested Partition Tags</td>
<td>58</td>
</tr>
<tr>
<td>4-4</td>
<td>Example of READFULL Tag</td>
<td>60</td>
</tr>
<tr>
<td>4-5</td>
<td>Example of WRITETOALL Tag</td>
<td>61</td>
</tr>
<tr>
<td>4-6</td>
<td>Example of PRML Tags</td>
<td>62</td>
</tr>
<tr>
<td>5-1</td>
<td>Parser for PRML</td>
<td>64</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 3-1 Analogy between VLDB and Call Center ................................................................. 35
Table 3-2 Transactions of TPC-E Benchmark ........................................................................ 43
Table 3-3 Footprints of TPC-E Benchmark Transactions ................................................... 44
Table 3-4 Initial TPC-E Table Sizes ..................................................................................... 45
Table 7-1 Examples of N_{0.5} and N_{0.25} .......................................................................... 85
Table 8-1 Experiment Result of Overhead Test ..................................................................... 106
Table 8-2 Experiment Result of TPC-E Benchmark Test ..................................................... 108
CHAPTER 1 INTRODUCTION

Database is a critical component in the present and future information processing systems. Its importance cannot be under-estimated. Very Large Scale Database (VLDB) is inevitable.

Unfortunately, database technologies fall short in meeting growing application demands, in particular, the challenges in performance scalability and service reliability.

According to David DeWitt and Jim Gray “the (ultimate) challenge is to build a high performance database system out of infinitely many processors of finite speed, and to build an infinitely large memory with infinite memory bandwidth from infinitely many storage units of finite speed [4].” For service reliability, the challenge is “building highly available database system out of infinitely many database servers of limited reliability”.

Meeting both high performance and high availability goals has been elusive since the beginning. The technical barrier has been the synchronous parallel transaction replication or the “update in place” problem – a “poisoned apple” by the most well-known researchers in the field [25].
1.1 Prior Work

There are two important recent breakthroughs in distributed parallel processing research:

a) Synchronous Parallel Replication (SPR) by Justin Y. Shi and Suntain Song; and
b) Stateless Parallel Processing (SPP) by Justin Y. Shi.

The synchronous parallel transaction replication problem was resolved using three techniques implemented as a runtime substrate between database clients and a cluster of shared-nothing database servers:

a) Dynamic transaction serialization when replicating transactions
b) Dynamic query load distribution.
c) Non-stop database resynchronization.

This runtime substrate (called DBx gateway) captures concurrent database transactions to perform above three functions. In addition, three application markup statements are devised to ensure the ACID properties of all concurrent transactions:

a) Lock/Unlock X. This is a mutex control statement that ensures serialization transactions with multiple concurrent updates on a critical segment X.
b) Load Balance On/Off. This is a load balancer switch that forces load distribution of enclosed transactions without serialization analysis.
c) Replication On/Off. This switch is designed for cluster administrative tasks. It is also useful nondeterministic database functions, such as getting a GUID or other values generated from database servers.
Dynamic serialization drastically reduces overall transaction replication overheads by serializing only concurrent update conflicts. In contrast, all existing replication methods require serialization of all transactions. Dynamic serialization drastically improves transaction replication speed.

Dynamic query distribution enables scalable performance leveraging multiple redundant synchronized datasets.

Non-stop resynchronization involves a non-trivial application sequence of traditional backup/restore routines resulting in a small constant service downtime (< 60 seconds) for resynchronizing arbitrarily large datasets. This step can be technically considered as the 2\textsuperscript{nd} phase of traditional two-phase-commit protocol.

These techniques have actually detoxed the “poisoned apple” as mentioned by Jim Gray[25]. They enable higher performance and higher availability at the same time for applications with heavy read-only queries. Performance degradation is expected for update-heavy applications.

Stateless Parallel Processing (SPP) is a technology pioneered by Justin Y. Shi for highly available high performance computing systems. SPP was inspired by the packet switching protocols for building large scale networks that require high performance and high reliability at the same time. The key concept is the “decoupling” principle that allows
spatial and temporal separation of concurrent events. Applying SPP to database cluster enables reliable parallel processing for data intensive transactional applications. The result is the Redundant Array of Independent Database (RAIDB) architecture.

RAIDB is a partitioned shift-mirrored database architecture that leverages data redundancy for parallel processing (see Figure 1-1). It allows parallel processing of select and update queries.

![Figure 1-1 RAIDB Conceptual Design](image)

1.2 Statement of Research Problem

Two prominent features are common among Very Large Scale Databases (VLDB): a) very large dataset; and b) very high transaction rate. Although higher level redundancy improves higher reliability and higher potentials in parallel processing, for practical applications a variable degree of redundancy can prove to be more appropriate.

Variable degree of redundancy can allow higher resource efficiency by reducing the replication overheads both in storage and in serialization time. Proper choice can enable
practical optimal configuration of VLDBs that can deliver cost efficient large scale data service with scalable high performance and scalable reliability at the same time.

The research problem is to identify mechanisms and theoretical performance and reliability impact using variable degree of redundancy.

1.3 Envisioned Contributions

This thesis presents a new automatic load balancing methodology for general-purpose transactional applications using multiple shared-nothing servers with variable degree of redundancy. The envisioned contributions of this research include:

1. Design and implementation RAIDB architecture with variable degree redundancy
2. Design and implementation of Functional Horizontal Partitioning method to provide non-stop dataset migration for RAIDB architecture
3. Design and implementation of Partition Replication Markup Language (PRML) to support dynamic transaction load balance.
4. Performance validation using industry standard benchmarks

1.4 Thesis Organization

This dissertation is organized as follows. Chapter-2 provides the background information. Chapter-3 presents the new Functional Horizontal Partitioning methodology. Chapter-4
describes the motivations for developing Extended Markup Language, and presents programming structures using PRML. Chapter-5 explains how to handle the PRML statements in pre and post processing. Chapter-6 discusses additional methodologies developed for the RAIDB system. Chapter-7 discusses performance models of the basic select/insert/update/delete operations on RAIDB system. Chapter-8 presents experiment results from TPC-E OLTP database benchmark with mixed transaction types. Chapter-9 contains the final conclusions and discussion about future work.
CHAPTER 2 BACKGROUND

Distributed database research involves a very large set of topics. This chapter presents only the materials that are relevant to database performance and reliability.

2.1 Transaction Load Balancing

The idea of transaction load balancing is to use multiple processors to share the work loads. Traditional mainframe computers have been proven too slow and too costly to catch up the speed of transaction increase in OLTP environments. Multiprocessors are more cost-effective systems for high performance transaction processing. However, technical progresses have been slow. A theoretically grounded multiprocessor database system should not contain shared objects (such as shared disks or memories), because the shared objects will become the performance bottlenecks and single point of failures. Ironically, all existing commercial database clusters share disks. The technical challenges of designing a distributed database architecture that can utilize the collective powers and resources of inherently unreliable servers and networks have not been met until recently.

There are two fundamental challenges: (1) how to cost-effectively maintain multiple synchronized copies of data; (2) how to provide optimal load distribution (parallel processing) leveraging the multiple synchronized data copies.
To keep multiple synchronized copies of data on shared-nothing servers, an efficient zero-loss transaction replication method is necessary.

2.2 Transaction Replication Technologies

2.2.1 Serial Replication Technologies

Existing transaction replication technologies can be put into two categories: asynchronous and synchronous.

Synchronous replication methods use Two-Phase-Commit protocol (2PC) to replicate transactions. All nodes must agree on committing a transaction submitted by a coordinator. Failures on any node will rollback the whole transaction. Two-Phase-Commit is a blocking protocol, servers are blocked serially for each transaction. The best cluster performance is determined by the slowest server in the cluster. The overall availability of the system is less than a single node since any node failure necessitates a planned downtime for recovery. Serial synchronous replication delivers two benefits: reduced unplanned downtimes and lossless transaction. It suffers increased planned downtimes and severe performance degradation.

Asynchronous replication technology is designed to reduce the waiting time by eliminating 2PC. It maintains a master server for each data object. Any write operation (insert, update and delete) must be processed on the master node first; the master node then transmits the
data changes serially to other nodes. Transaction log is commonly used for this purpose [7]. The master node must be throttled below the serial replication speed in order to avoid replication queue overflow. Serial asynchronous replication technology delivers reduced unplanned downtime and a delayed copy of replicated data. Therefore, it cannot guarantee lossless transactions. It also suffers increased planned downtimes and performance degradation due to master throttling.

### 2.2.2 Parallel Replication Technology

Parallel replication has been considered a “poisoned apple” [25]. It touches upon many fundamental problems in distributed database architecture research. Managing the states of multiple concurrent transactions is the root cause for transaction replication difficulties.

A recent breakthrough in parallel synchronous replication (PSR) provides a practical solution to the serial replication problems. This breakthrough came after the recognition of the fact that the only stateless moment of a transaction is when it is in transit [1]. With the capability to manage multiple concurrent stateless transactions, PSR enables

a) **Dynamic serialization**: serializations are only needed when replicating concurrent update conflicts.

b) **Dynamic load balancing**: read-only queries can be evenly distributed to boost performance.
To maintain multiple concurrent stateless transactions, one must develop a runtime substrate between database users and a cluster of shared-nothing database servers. In other words, a network middleware gateway is needed to capture all in-flight concurrent transactions. DBx is one such product developed by Justin Y. Shi and his student Suntian Song.

Since the DBx gateway can only “see” in-flight transactions, to ensure the atomic property for transactions with multiple updates, the application needs to markup concurrent updates with proper MUTEX – a discipline well practiced in multi-threading programs.

The critical network position of the DBx gateway can also solve the difficulties in 2PC. It can deliver a magical “half-twist” when resynchronizing multiple out-of-sync database servers with worst-case 60 second service downtime, independent of data size.

Since each server node in a DBx cluster works independently and all datasets are kept synchronized at all times, failover is in real time. Each node maintains full control of local resources. This autonomy protects the server from interference of others.

The DBx cluster offers four additional benefits:

a) Lossless transaction processing: Each transaction can only be in one of three states: success, failure or timeout. As long as each timeout is treated with a correct re-try logic (containing a commit uniqueness check), no transaction can be lost.
b) Zero single-point-of-failure: The lossless feature allows a DBx cluster be protected by multiple passive stateless DBx gateways.

c) Horizontally scalable: Multiple DBx gateways can be deployed to distribute the replication work loads for different applications.

d) Byzantine failure prevention: DBx cluster requires the designation of a “primary server”. The clients’ computation states are kept consistent with the “primary server” at all times. This design along with non-stop resynchronization prevents Byzantine failures.

Before the PSR technology, the degree of redundancy is typically two (2) to avoid the costly overheads of transaction replication. With PSR technology, redundancy helps with performance. PSR enables scalable performance and reliability by adding redundancies.

DBx can deliver scalable performance and availability for heavy-read applications. Performance degradation is expected for heavy-write applications.

2.3 Data Partitioning

Data partitioning is widely used in databases to improve transaction processing performances. It divides logical database objects (tables) into multiple independent divisions. These divisions (partitions) usually reside on separated disks or servers to gain optimal performance.
Partitioning is also used to simplify programming and administration tasks. In modern database systems, such as Microsoft SQL Server and Oracle, programmers do not need to modify their SQL DML statement to obtain the performance benefits of data partitioning.

Tables can be partitioned along rows (horizontal partition) or columns (vertical partition). The partitioned datasets may reside on a single database server (internal partition) or multiple shared-nothing servers (external partition). To hide the complexities, Partitioned View and Partitioned Table are used.

2.3.1 Horizontal Partitioning and Vertical Partitioning

Horizontal external partition (HEP) puts a table row-wise onto multiple independent server nodes. All servers have the same schema but with smaller datasets. It gains performance by executing client queries on smaller dataset. A large query is translated into multiple sub-queries plus some post-processing work. HEP is supported in Microsoft SQL Server, Sybase, Oracle, IBM DB2, MySQL etc.

HEP configuration is maintained on each individual server. It is not possible to add new nodes or adjust the partition distribution without shutting down the cluster. The overall availability of a HEP cluster decreases exponentially as the number of nodes increases in the cluster because any node crash will prevent the cluster from working properly.
Vertical external partition (VEP) distributes a table column-wise to multiple shared-noting
servers. In this architecture, tuples of relation are partitioned so that multiple share-noting
nodes can access their local devices in parallel. VEP gains performance by distributing the
processing column-wise to multiple servers. Each server has a different schema. To get the
result for a single query, multiple processors must get involved to scan their local device.
This causes a lot of exchanged messages. Besides, the Post-processing in VEP is more
complex than HEP since it must deal with multiple different schemas. For this reason, we
focus only on HEP in this dissertation.

There are also a number of experimental systems focused on replicated partitioned
databases.

Lasaro Camargos and his colleagues proposed a partitioned IMDB (In Memory Database)
Sprint system [17]. The Sprint system uses multiple servers to host partitioned tables to
process in local memories. Disk writes are handled by special Durability Servers in strict
sequential manner. The Durability Server is replicated in serial to provide failure resilience.
Transaction replication in Sprint uses strict serial multicasts. Each data server casts its disk
write request to the Durability Servers. Durability Server act as coordinator that commits
the update to disk in strict serial order.

Tests results reported by Camargos shows that “Experiments abort rates are highly
dependent on the percentage of global transaction (access data on multiple servers) in the
workload, up to 25% of aborts when all transaction are global” [17]. The strict serial
replication orders are the root cause for these aborts (timeouts). While the system can execute very high read-only queries (in memory), the overall system is not scalable.

The similar IMDB architecture can also be found in GORDA [3]. Xiaohu Yang builds dynamic data partitions by loading part of data tables into local server memory [9]. The partition algorithm is carefully selected so that each node keeps different piece of data to avoid contention. Yang’s design removed the transaction replication to eliminate synchronization overheads (serialized multicast). It postpones transaction replication to Durability Servers. This process is serial; the high performance penalty cannot be avoided.

Using PSR can drastically reduce serial replication overheads to solve the system availability problem in partitioned servers. Figure 2-1 shows a database cluster using PSR with horizontal partition. Each sub-gateway represents a single replicated partition.

![Figure 2-1 PSR on Traditional Horizontal Partitioning](image-url)
2.3.2 Internal Partitioning and External Partitioning

All existing database horizontal partition architectures can be grouped into two categories, internal partition and external partition.

Internal partition is a method of partitioning data onto multiple disks controlled by the same server. Since disks are inherently slower than processor and memory, parallel disk accesses can deliver performance speedup [10]. The advantage of internal partition is that it doesn’t need exchange message with other servers.

Internal partitioning can be implemented by partitioned tables or partitioned views. In a database server system with multiple disks, internal partitioning allows maximizing single node performance.

External partitioning is a method of partitioning data onto multiple shared-nothing server nodes (see Figure 2-2). It is usually implemented via partitioned views. Each server node harbors part of the full dataset [5], [13], and [16]. External partition has its own shortcomings. With data distributed among multiple nodes, the availability of overall VLDB system is lower than any single member node. Therefore, it suffers exponentially decreasing service availability problem.
Data skew and adaptive data migration are two major challenges for partitioned databases [11], [19]. The distribution of data and load may vary dramatically as the data size grows. It is difficult to predicate the actual transaction distribution change when the data partition schema is designed. Adaptive data distribution is necessary to correct the data skew and to move data to other server nodes [13], [14]. To date, there have been no accepted standards to meet these requirements.

Cluster expansion and contraction is another challenge for partitioned database cluster. Application developers must reconfigure/rewrite code to adjust the new partition distribution. This is laborious and error-prone.
2.3.3 Partitioned View and Partitioned Table

Programming with partitioned database schema can become very complex. Partitioned View and Partition Table are designed to ease this task.

Partitioned view is first introduced in Oracle7, and it was quickly adopted by other database vendors, such as MS SQL Server, IBM DB2 Universal Database (DB2 UDB), Sybase and MySQL etc. Figure 2-3 gives an example of partitioned view.

```
Create view vwCustomers
As
Select * from Server1.MyDB.dbo.CustomerA
Union All
Select * from Server2.MyDB.dbo.CustomerB
Union All
Select * from Server3.MyDB.dbo.CustomerC
```

Figure 2-3 Example of Partitioned View

Figure 2-3 shows that data in the logical database object vwCustomers is a union of all records from three different servers. It gives the application developers the convenience to use a single view while allowing scaling out to use more servers when necessary.

To take advantage of the automatic parallel processing feature of partitioned view, all servers referenced in partitioned view must be connected as linked server. These linked servers must be created before the creation of the partitioned view.
When a user queries vwCustomers in Figure 2-3, the hosting SQL Server forwards the query to all linked servers referred in the partitioned view and merges the returned dataset before sending back to the user. Partition view relies on all the linked servers referred by its definition. The Partitioned View falls apart if any linked server crashed. This dependency affects the performance and availability of the entire system.

Partitioned View also has restricted usage. For example, in MS SQL Server 2005, “INSERT, UPDATE, and DELETE actions against a partitioned view are not allowed if there is a self-join with the same view or with any of the member tables in the statement. In Oracle11g, “A DML operation cannot be parallelized if it is in a distributed transaction or if the DML or the query operation is against a remote object.[24]” Because of these restrictions, partitioned views are usually called “pseudo” partitioning among DB developers. These restrictions make Partitioned Views difficult to use.

Partitioned tables relieve all the restrictions on partitioned views. It is called “native” partitioning among DB developers. Partitioned Tables are supported by most database systems, including MS SQL Server 2005, Oracle11g, Sybase, IBM DB2 and others. Partitioned tables support all the properties and features associated with designing and querying standard tables and indexes, including constraints, defaults, identity values. It simplifies the administration tasks and improves performance of the hosting database server.
Although both vertical and horizontal partitioning can be manually implemented locally or across servers, due to the lack of proper replication technology, Partitioned Table is typically used for horizontal partitioning within a single server.

### 2.3.4 Partitioning Criteria

Partitioning criteria define the data distribution strategy. There are five most commonly used partitioning criteria:

- **Round Robin partitioning**
  Round Robin partitioning is the simplest partitioning strategy. Rows are assigned to each partition in a round-robin manner to achieve load balancing.

- **Range partitioning**
  In each range partition, the partition key values are in a predefined range. For example, we can use [transaction date] column to partition a transaction table by each month.

- **Hash partitioning**
  Hash partitions use a hash algorithm on the partitioning columns to decide the partition assignment for each row. The selection of hash function and the partition key determines whether the dataset is evenly distributed or skewed distributed.

- **List partitioning**
  If the values of the partitioning column are predefined, one can create a partition for each value using the list partitioning method. List partitioning is good for managing partitions using discrete values.
• **Composite partitioning**

  This method combines the methods of the above partitioning schemes. For example, we can first use a range partitioning and then a hash partitioning to create a composite partitioning schema.

### 2.4 Parallel Replicated Partitioned DBMS: RAIDB

Stateless Parallel Processing (SPP) is a technology pioneered by Justin Y. Shi for highly available high performance computing systems. SPP was inspired by the packet switching protocols for building large scale networks that require high performance and high reliability at the same time. The key concept is the “decoupling” principle that allows spatial and temporal separation of concurrent events. Applying SPP to database clusters enables reliable parallel processing for data intensive transactional applications. The result is the Redundant Array of Independent Database (RAIDB) architecture.

RAIDB is a conceptual design about partitioned shift-mirrored database architecture that leverages data redundancy for parallel processing (see Figure 1-1). Similar to the RAID storage systems, RAIDB allows parallel processing of both read-only and data-change queries. An additional benefit is that RAIDB allows replication and load balancing using remotely connected replicas. Currently, the RAIDB structure can only be implemented manually. It is understood that keeping full data redundancy on each server gives the best read-only parallel processing capability, but it is gives the worst-case update overhead.
2.5 Markup Language for Transaction Replication Technologies

Markup language is widely used to inject controls in the target document or program. XML, HTML and SGML are three well known markup languages. According to the Taxonomy Guide by the Professional Learning Center of University of Toronto, “*Markup language is a set of codes or tags that surrounds content and tells a person or program what that content is (its structure) and/or what it should look like (its format). Markup tags have a distinct syntax that sets them apart from the content that they surround.* [22]”. Markup languages are typically extensible to allow easy change and additions.

Markup languages can be used to manage parallel transactions by injecting very complex controls, such as OpenMP [26], PML (Parallel Markup Language) [20] and RML (Replication Markup Language, also known as Embedded Statements) [1].

In this thesis, we will extend RML to describe the dependency patterns by marking partition oriented program segments. With the new extension for partition replication, we call the new markup language as PRML (Partition Replication Markup Language). The existing RML will be a subset of the new PRML.

2.5.1 RML Overview

RML statements are inserted into a database application as SQL comments. This design avoids the statements be interpreted by SQL parser. They are to be interpreted by the runtime substrate DBx gateway.
RML statements help programmer achieving optimal application performance by providing the following controls [1].

1. *Fine grain dynamic locking for data consistency and the optimal serialization performance.* (ICXLOCK ON Name Level, ICXNOLOCK)

2. *Replication engine control for server-side functions that only need to run on any single server.* (ICXNR on and ICXNR off)

3. *Load balancing engine control for complex queries and stored procedures.* (ICXLB on and ICXLB off)

An RML tag includes the following parts

- Header
- Token tag
- Token switch (switch tag) or LockID Lock_Level (lock tag)

For example, “--SET ICXLB ON” and “--SET ICXLB OFF” are one pair of RML tags for controlling the load balance of enclosed transactions. Each query wrapped between these two tags is evenly distributed to one and only one single server. In these two tags, “--SET” is the tag header; “ICXLB” is token tag; “ON” and “OFF” are the token switches. Unlike XML, empty tag is not permitted in RML.

A RML tag must be sent as a whole to be correctly executed. Each tag must have a corresponding closing tag. Figure 2-4 gives an example using two pairs of nested RML tags. One pair of RML tags (ICXLOCK) is nesting in another pair of RML tags (ICXNR).
Applications with RML tags are highly portable. They are executable on any database platforms with or without DBx gateways.

2.5.2 RML Parser

The RML parser is designed to interpret all RML tags submitted by client application. Figure 2-5 gives a simplified flow chart about the nesting control in RML. The parser and tags work well for PSR system with full data redundancy on each server. They help programmer to reduce the transaction isolation level from table level to row level. They also help programmer to achieve optimal performance for both read-only queries and write operations. RML statements are very helpful for developers to exercise controls of the replication and load balancing engines.
2.5.3 New Extension on Partitioning - PRML for RAIDB System

This dissertation intends to extend the capability of RML to support RAIDB systems using data partitioning. This involves new tag design and implementation of a new RAIDB gateway. We call this new markup language as **Partition Replication Markup Language** (PRML).
To support RAIDB systems using data partitioning, the data partition information must be properly maintained among application, DBx gateway and background RAIDB cluster.

With the presentence of variable redundancy degrees, integrating distributed data partitions with transaction replication and information retrieval gets more complicated, because it involves different schema on multiple shared-nothing servers now. The parser of RML must be extended to deal with the non-homogeneous data distribution.

Since RAIDB gateway sits between clients and database servers, it doesn’t have any database information except for passing SQL statements. The RAIDB gateway will need additional information to determine the correct location of the target data. New tags for PRML are needed to wrap such information and transfer it to the RAIDB gateway, which will retrieve this information at runtime. Using such partition information allows RAIDB gateway to hide the partitioning complexities by automating the pre and post processing. In particular, data records from multiple partitioned tables can be locked by a single partition key. This feature provides an additional layer of object control between table lock and row lock, thus providing means for maximal concurrency for the best performance. We will discuss these new PRML tags in chapter 4.

2.6 Summary

The PSR technology solves the high speed lossless transaction replication problem. It doesn’t require any message exchange among member servers. It provides the basis for
building fault resilient high performance RAIDB system. Currently, PSR technology replicates every write operation to all member nodes, which make it fit well for read intensive VLDB system. To deliver high performance for write intensive database, a higher level of load distribution is needed to balance both read-only and write operations in addition to PSR current capabilities. This dissertation will develop a new customizable data partitioning technology to overcome those problems experienced by researchers.
CHAPTER 3 DESIGN OF RAIDB WITH VARIABLE DEGREE OF REDUNDANCY

Speedup and scaleup are two key properties for an ideal VLDB system. Linear speedup represents N-times as much hardware can perform a task in 1/N the elapse time. Linear scaleup represents N-times as much hardware can perform N-times as large a task in the same elapsed time. For a very large OLTP system with an extremely large number of transactions per second, the measurements for these two properties can be replaced by one performance measurement: throughput speedup. We define throughput speedup (Sp) for a database system as following

\[
Sp = \frac{T_1}{T_p}
\]

where:

- \(T_1\) is the execution elapse time of a bunch of (concurrent) transactions on an OLTP database system with single server.
- \(T_p\) is the execution elapse time of a bunch of (concurrent) transactions on an OLTP database running on a RAIDB system.
- \(p\) is the number of independent servers in the RAIDB system.

For a very large OLTP system, throughput speedup is a much desired property. The driving force behind throughput speedup is the demands of applications that have to process an extremely large number of transactions in unit time. In this chapter, we present a new partition methodology to provide high throughput speedup for very large OLTP database
system. We will use the throughput speedup defined in Eq. 3-1 throughout our discussion in this thesis.

Throughput and response time and are two related performance measurements for a database system. Throughput represents the number of tasks that can be completed in a given time interval. Response time represents the time interval since a single task been submitted until the result been delivered. If the average response time of all transactions gets shorter, the overall throughput will increase. It works the same vice versa.

Like every system designed by human being, there is an upper limit on the throughput for every OLTP database system. The response time of each individual task will get longer and longer when the total number of task exceeds the maxim allowed value. All concurrent tasks will compete for limited resource and processor power. The system will be kept busy on context switch and cannot do any real work. The whole system throughput deteriorates quickly. It is usually called thrashing (shown in Figure 3-1). Such thrashing can even happen earlier for a VLDB with multiple synchronized data copies, if servers need exchange message with each other and maintain full synchronized data copy on every host. Figure 3-1 shows a thrashing phenomenal for a typical OLTP system.
3.1 Challenges and Expected Output

As the sizes of datasets and transaction load keep on growing, the simple strategy of load-balancing read-only queries and replicate write operations to all nodes cannot catch up with the transaction grow speed. The cost of keeping full redundancy data on each server is getting more expensive as the data size grows. It is necessary to balance the load of write operations too. A new data partitioning architecture is needed for VLDB system using
parallel synchronized transaction replication technology. This proposed research must answer the following challenges:

a) How to provide high data availability (redundancy) without sacrificing system performance?
b) How to provide an application programming interface (API) that requires minimal re-coding efforts to achieve optimal performance?
c) How to conserve resources (memory, disk space and network bandwidth) with quantified compromises in performance and reliability?
d) How to maintain fault resilience without shutting down the service when node is added or removed?

A VLDB system satisfying above challenges shall be able to complete clients’ request within reasonable elapse time, even as the size of the database and the number of transactions keep on growing.

The scalability property of an ideal VLDB system should like the one in Figure 3-2. With N-times resource, the overall system throughput should be N-times as the original system. The thrashing of such VLDB should appear N-times later than the original system too.
This chapter will propose a new partitioning methodology to solve the scalability problem for VLDB using parallel synchronous transaction replication architecture. This new methodology will allow automatic data/transaction distribution and seamless (non-stop) recovery.

Figure 3-2 Expected Performance Gain of Shared-Nothing Cluster
3.2 TPC-E Example Database

To demonstrate the procedure of creating and deploy of the new data partitioning technology on a VLDB system, we will use a sample database to demonstrate the analyze work and design logic. The sample database is adapted from the On-Line Transaction Processing (OLTP) benchmark workload TPC-E, which is developed by Transaction Processing Performance Council (TPC) [18]. This database schema is designed to represent modern OLTP database systems. It emulates a brokerage house's real time internal business processing. Data of customer, broker, security, trade information are stored in this database. Figure 3-3 gives a simplified E-R graph of the database schema.

**Figure 3-3 E-R Diagram of TPC-E Benchmark Database**
3.3 Schema Design of VLDB with Variable Degree of Redundancy

To make the requirement for partitioning architecture in VLDB system be easily understood, I will use an example of a customer call center. Customer call center CCC helps customer track their orders or make orders over telephone by answering customers’ calls. The call center manager hires agents to answer these phone calls. There are two kinds of agent, junior agent and senior agent. Junior agents have limited privilege and can only access part of the whole products carried by this center. Different junior agent may be in charge of different products. Each agent keeps a synchronized list of the products. The data on the list are always accurate, although some junior agents’ lists may be incomplete. Senior agents have full privileges on all products and have a full copy of all synchronized data. A junior agent may be promoted to be a senior agent after proper training. A senior agent may be demoted to a junior agent if he/she loses some privileges. When customers call in, they must go through an automated menu to forward their calls to the agent(s) who can fulfill their requests. If the phone call is just to track an existing order or to get a price quote (read-only query), such question can be answered by a single agent who has the proper information about such queries. If the customer wants to place a new order about a specified product, such request will be broadcasted to each agent who are in charge of that product, so that all these agents can maintain consistent copies and know what is going on between that customer and CCC. For efficiency purpose, each agent should be able to work independently and know the business rules and procedures he/she has been trained to use. Customers make phone calls to the center for service. They don’t care who answered their calls or how many customer agents are working in that center, as long as their concerns are
fulfilled quickly and processed correctly. Senior agent can take the spot of any junior agent what that junior agent is absent. The manager can adjust the pool of agent basing on the volume of customer phone call. To make sure the call center can serve customer 24x7, the manager must maintain the agent staff at a level that it can answer all kinds of customer queries at any moment. If there is only one agent on duty, that agent must be a senior agent, who knows all the rules and procedures. The structure of above call center is shown in Figure 3-4.

**Figure 3-4 Business Model of Custom Call Center**

VLDB systems should work like above customer call center. The fundamental element of a VLDB system is database server node. One VLDB system may include one or more nodes.
Each server should be self-reliant and independent from the status of other nodes. A health server node should be able to process the load assigned to it quickly and correctly. When a member server is removed from the cluster (due to hardware/software crashes, system maintenance, database rebuild and resynchronization, etc), the rest active nodes should be able to handle all queries quickly and correctly. If there exists only one health active node in the VLDB system, that node should have a full copy of all data records to guarantee full data availability and full functionality service. We will call such server with full data copy as **Full Server**. We call a server with only part of the full data set as **Slim Server**. The full server should be optional for a VLDB system, even though it provides many conveniences for query optimize and data resynchronization. In an update-intensive system, full server node may become the bottleneck of the whole system because all write operations must be replicated to such nodes.

<table>
<thead>
<tr>
<th><strong>Customer Call Center</strong></th>
<th><strong>VLDB System</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>Database client</td>
</tr>
<tr>
<td>Automated menu</td>
<td>Middleware Gateway to forward load to its corresponding server node(s)</td>
</tr>
<tr>
<td>List of products and order records</td>
<td>Data set</td>
</tr>
<tr>
<td>Part of the products list and order records</td>
<td>Partition of Dataset</td>
</tr>
<tr>
<td>Agent</td>
<td>Database Server Node</td>
</tr>
<tr>
<td>Junior agent: people who keeps a synchronized incomplete records</td>
<td>Slim Server: database server node keeps part of overall dataset</td>
</tr>
<tr>
<td>Senior agent: people who keeps a synchronized full records</td>
<td>Full Server: database server node keeps a full copy of overall dataset</td>
</tr>
</tbody>
</table>

**Table 3-1** summarizes the analogy between VLDB and the call center mentioned above.

---

35
3.3.1 Outline of VLDB with variable redundancy

Figure 3-5 displays one VLDB system using our new data partition architecture. There are 3 slim servers and one full server in that system. The schema of all partitions is maintained on each node. Each node works independently and is self-reliant. It avoids any communication among member node. It guarantees that partitioned data set can be migrated easily to any selected thin node and be promoted as a full server. Vice versa, the way to demote a full server node to a slim server node is trivial.

The virtual table and PSR gateway in Figure 3-5 provide two layers of information encapsulations. PSR gateway encapsulates multiple server nodes into a VLDB system. The virtual table encapsulates those real partitions as a virtual table. Since each database server is working on a complete partition schema on its own storage, they can theoretically work independently.
Figure 3-6 gives a brief procedure to create a VLDB supported by PSR middleware gateway using variable data redundancy.

1. Analyze all transactions to be executed on the target database. Pay attention to those tables accessed by intensive write operation.

2. Analyze the dependency among tables (see detail in section 3.3.3). Select proper target table(s) for load balance purpose.

3. Choose partitioning criteria and partitioning function.

4. Split the selected table(s) into multiple data partitions. Replace the original table(s) by virtual table(s) (see Figure 3-7)

5. Embed Partition Replication Markup Language (PRML) statements into application code for optimal performance. (We will give more detailed description about PRML in Chapter 4)

6. Migrate those partitioned data to other database servers. Add and activate these new server nodes in the RAIDB configuration (see Figure 3-8).

Figure 3-6 Procedure of Functional Horizontal Partitioning Implementation
Figure 3-7 Partition Data whining Single Server

Figure 3-8 Partitioned Data Migrate
The partition replication procedure in Figure 3-8 is very similar to the Data Publisher/Subscriber architecture supported by many commercial database systems, shown in Figure 3-9. However, there exist many differences between these two architectures.

- First, Publisher/Subscriber architecture uses asynchronous transaction replication method. There exists a queue for each subscription to store new transaction records. RAIDB system uses synchronous replication technology, and the partition migration will stop after the partition is resynchronized on new server(s).
• Second, the transaction queues for Publisher/Subscriber architecture consume resource on the publisher and it will be the bottleneck for a busy system. The performance of a RAIDB system will not be limited by such transaction queues.

• Third, the only full and up-to-date data copy is only maintained by the publisher in a Publisher/Subscriber architecture. If the publisher crashes, all the data in those transaction queues will get loss. The availability of the whole system Publisher/Subscriber architecture cannot be greater than the single publisher. Those subscribers cannot guarantee zero transaction loss if the publisher crashes. In a RAIDB system, each active server node keeps an up-to-date data copy. As long as there exists an up-to-date active data copy for each partition, the RAIDB system can continue service or rebuild a full server if necessary. Adding more hardware in RAIDB does provide higher availability.

• Fourth, the subscribers in Publisher/Subscriber architecture can only load balance some non-critical read query. All critical write and read operations must first go through the publisher. It cannot deliver real load balance. In a RAIDB system, both read and write operations can be maximally distributed among all member nodes.

3.3.2 Data Partitioning for RAIDB system using PSR Middleware gateway

To choose the partition candidate(s) for PSR middleware architecture, all transactions executed on the target database need be analyzed. The footprints of each transaction need
be tracked. Each table needs be evaluated base on its size, relationships with other tables and the frequency and pattern accessed by each transaction.

Tables in an OLTP relational database can be put into three types basing on size and the transactions pattern accessed. These three table types are: fixed size table, scaling table and growing table.

The size of a fixed size table is very stable and its content doesn’t change frequently in daily work. A table listing all industries or the zip code of every county in United States serves good examples for such fixed table. Almost all queries on such tables are read-only queries. Therefore, such tables are perfect candidate to be replicated on each server node. They just need be replicated once on each server to balance read-only queries. These tables usually don’t need be partitioned across server members (external partition). If the sizes of such tables are too big, using index or internal partition can effectively reduce the query response time on such tables.

The size of scaling table usually depends on the size of some key table. For example, in the TPC-E database schema the total row count of CUSTOMER_ACCOUNT table depends on the row count number of CUSTOMER table. One customer may have zero or multiple accounts, but a customer’s total account number doesn’t keep on changing everyday. On the other hand, the size of customer table may keep on growing at a moderate rate. Tables in this category can be partitioned by our new partitioning methodology if write operations on these tables are very intensive.
As the name suggested, growing tables are tables with initial cardinalities to one key table, such as customer table. The size of growing table keeps on growing as more transactions are executed on the database. TRADE HISTORY tracking the information of every trade is such kind of table. Such growing tables need be carefully designed and deployed. It consumes a lot of system resource and deteriorates overall system performance when the size gets bigger and bigger. These tables are good candidate for data partitioning.

**Listing of Tables in TPC-E Benchmark**

<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>BROKER</th>
<th>MARKET</th>
<th>DIMENSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNT_PERMISSION</td>
<td>CHARGE</td>
<td>EXCHANGE</td>
<td>STATUS_TYPE</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>COMMISSION_RATE</td>
<td>INDUSTRY</td>
<td>TAXRATE</td>
</tr>
<tr>
<td>CUSTOMER_ACCOUNT</td>
<td>TRADE_TYPE</td>
<td>SECTOR</td>
<td>ZIP_CODE</td>
</tr>
<tr>
<td>CUSTOMER_TAXRATE</td>
<td>BROKER</td>
<td>COMPANY</td>
<td>ADDRESS</td>
</tr>
<tr>
<td>WATCH_ITEM</td>
<td>CASH_TRANSACTION</td>
<td>COMPANY_COMPETITOR</td>
<td></td>
</tr>
<tr>
<td>WATCH_LIST</td>
<td>SETTLEMENT</td>
<td>DAILY_MARKET</td>
<td></td>
</tr>
<tr>
<td>HOLDING_HISTORY</td>
<td>TRADE</td>
<td>FINANCIAL</td>
<td></td>
</tr>
<tr>
<td>HOLDING</td>
<td>TRADE_HISTORY</td>
<td>LAST_TRADE</td>
<td></td>
</tr>
<tr>
<td>HOLDING_SUMMARY</td>
<td>TRADE_REQUEST</td>
<td>NEWS_ITEM</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NEWS_XREF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SECURITY</td>
<td></td>
</tr>
</tbody>
</table>

Legend:

- **Fixed table**
- **Scaling table**
- **Growing table**
- **Growing/Scaling table**

Figure 3-10 Categories of TPC-E Tables
Figure 3-10 shows the fourth table type: growing/scaling table. Actually this is not a new
table type. These tables are frequently accessed by transactions. A lot of rows will be
inserted into and/or deleted from these tables. These tables should be treated as growing
table, although the size of these tables may not increase as fast as other growing table.

Table 3-2 lists all transactions to be executed on the TPC-E benchmark database.
Transactions from 1 to 10 compose more than 99% transactions during benchmark test.
Transaction 11 and 12 are for data maintenance or cleanup purposes, these two transactions
compose less than 1% of overall transactions. The frequency of each transaction can be
adjusted for different test purpose.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Frequency</th>
<th>Access</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Broker-Volume</td>
<td>4.9%</td>
<td>Read-only</td>
<td>DSS-type medium query</td>
</tr>
<tr>
<td>2. Customer-Position</td>
<td>13%</td>
<td>Read-only</td>
<td>“What am I worth?”</td>
</tr>
<tr>
<td>3. Market-Feed</td>
<td>1%</td>
<td>Read-write</td>
<td>Processing of Stock Ticker</td>
</tr>
<tr>
<td>4. Market-Watch</td>
<td>18%</td>
<td>Read-only</td>
<td>“What’s the market doing?”</td>
</tr>
<tr>
<td>5. Security-Detail</td>
<td>14%</td>
<td>Read-only</td>
<td>Details about a security</td>
</tr>
<tr>
<td>6. Trade-Lookup</td>
<td>8%</td>
<td>Read-only</td>
<td>Look up historical trade info</td>
</tr>
<tr>
<td>7. Trade-Order</td>
<td>10.1%</td>
<td>Read-write</td>
<td>Enter a stock trade</td>
</tr>
<tr>
<td>8. Trade-Result</td>
<td>10%</td>
<td>Read-write</td>
<td>Completion of a stock trade</td>
</tr>
<tr>
<td>9. Trade-Status</td>
<td>19%</td>
<td>Read-only</td>
<td>Check status of trade order</td>
</tr>
<tr>
<td>10. Trade-Update</td>
<td>2%</td>
<td>Read-write</td>
<td>Correct historical trade info</td>
</tr>
<tr>
<td>11. Data-Maintenance</td>
<td>Once every 60 sec</td>
<td>Read-write</td>
<td>Time Triggered</td>
</tr>
<tr>
<td>12. Trade-Cleanup</td>
<td>Once every run</td>
<td>Read-write</td>
<td>Run once before Test Run</td>
</tr>
</tbody>
</table>

Table 3-2 Transactions of TPC-E Benchmark
Table 3-3 shows the footprints of the 12 transactions listed in Table 3-2. This table shows that, except the infrequent data-maintenance transaction, more than 90% read-write transactions access eight tables, which are highlighted in Table 3-3. Most transactions only access a few closely related records in these tables. These eight tables are good candidates for the partitioning strategy we proposed in Figure 3-6.

<table>
<thead>
<tr>
<th>Category</th>
<th>Table Name</th>
<th>Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>ACCOUNT_PERMISSION</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>CUSTOMER</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>CUSTOMER_ACCOUNT</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>CUSTOMER_TAXRATE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>HOLDING</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>HOLDING_HISTORY</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>HOLDING_SUMMARY</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>WATCH_ITEM</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>WATCH_LIST</td>
<td>RO</td>
</tr>
<tr>
<td>BROKER</td>
<td>BROKER</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>CHARGE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>COMMISSION_RATE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>CASH_TRANSACTION</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>SETTLEMENT</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>TRADE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>TRADE_HISTORY</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>TRADE_REQUEST</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>TRADE_TYPE</td>
<td>RO</td>
</tr>
<tr>
<td>MARKET</td>
<td>COMPANY</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>COMPANY_COMPETITOR</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>DAILY_MARKET</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>EXCHANGE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>FINANCIAL</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>INDUSTRY</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>LAST_TRADE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>NEWS_ITEM</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>NEWS_XREF</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>SECTOR</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>SECURITY</td>
<td>RO</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>ADDRESS</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>STATUS_TYPE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>TAXRATE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>ZIP_CODE</td>
<td>RO</td>
</tr>
</tbody>
</table>

[Note]: RO = Read-Only, RW=Read-Write
### Initial sizes of tables in TPC-E

<table>
<thead>
<tr>
<th>Table name</th>
<th>Symbol</th>
<th>Initial Table size of Partition Tables, byte</th>
<th>Initial Table size of Non-Partition Tables, byte</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trade_Request</td>
<td>TR</td>
<td>0</td>
<td></td>
<td>Growing table</td>
</tr>
<tr>
<td>Status_Type</td>
<td>ST</td>
<td>74</td>
<td>1</td>
<td>Fixed table</td>
</tr>
<tr>
<td>Trade_Type</td>
<td>TT</td>
<td>99</td>
<td>2</td>
<td>Fixed table</td>
</tr>
<tr>
<td>Charge</td>
<td>CH</td>
<td>180</td>
<td>3</td>
<td>Fixed table</td>
</tr>
<tr>
<td>Sector</td>
<td>SC</td>
<td>205</td>
<td>2</td>
<td>Fixed table</td>
</tr>
<tr>
<td>Exchange</td>
<td>EX</td>
<td>379</td>
<td>4</td>
<td>Fixed table</td>
</tr>
<tr>
<td>Broker</td>
<td>B</td>
<td>2,703</td>
<td>6</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Industry</td>
<td>IN</td>
<td>2,766</td>
<td>8</td>
<td>Fixed table</td>
</tr>
<tr>
<td>Commission_Rate</td>
<td>CR</td>
<td>6,240</td>
<td>12</td>
<td>Fixed table</td>
</tr>
<tr>
<td>TaxRate</td>
<td>TX</td>
<td>17,039</td>
<td>19</td>
<td>Fixed table</td>
</tr>
<tr>
<td>News_XREF</td>
<td>NX</td>
<td>83,893</td>
<td>100</td>
<td>Scaling table</td>
</tr>
<tr>
<td>WatchList</td>
<td>WL</td>
<td>115,000</td>
<td>117</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Last_Trade</td>
<td>LT</td>
<td>152,275</td>
<td>154</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Customer_Taxrate</td>
<td>CX</td>
<td>160,140</td>
<td>162</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Company_Competitor</td>
<td>CP</td>
<td>195,000</td>
<td>196</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Zip_Code</td>
<td>ZC</td>
<td>298,742</td>
<td>301</td>
<td>Fixed table</td>
</tr>
<tr>
<td>Address</td>
<td>AD</td>
<td>353,476</td>
<td>354</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Security</td>
<td>S</td>
<td>523,002</td>
<td>523</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Company</td>
<td>CO</td>
<td>523,930</td>
<td>524</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Customer</td>
<td>C</td>
<td>827,681</td>
<td>827</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Account_Permission</td>
<td>AP</td>
<td>1,692,445</td>
<td>1,692</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Customer_ACCOUNT</td>
<td>CA</td>
<td>1,944,070</td>
<td>1,944</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Holding_Summary</td>
<td>HS</td>
<td>5,995,052</td>
<td>6,000</td>
<td>Growing table</td>
</tr>
<tr>
<td>Financial</td>
<td>FI</td>
<td>6,942,122</td>
<td>6,942</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Watch_Item</td>
<td>WI</td>
<td>8,741,641</td>
<td>8,742</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Daily_Market</td>
<td>DM</td>
<td>180,840,910</td>
<td>180,841</td>
<td>Scaling table</td>
</tr>
<tr>
<td>News_Item</td>
<td>NI</td>
<td>501,915,895</td>
<td>501,916</td>
<td>Scaling table</td>
</tr>
<tr>
<td>Holding</td>
<td>H</td>
<td>304,843,599</td>
<td>304,843</td>
<td>Growing table</td>
</tr>
<tr>
<td>Settlement</td>
<td>SE</td>
<td>4,266,609,748</td>
<td>4,266,610</td>
<td>Growing table</td>
</tr>
<tr>
<td>Holding_History</td>
<td>HH</td>
<td>4,646,021,750</td>
<td>4,646,022</td>
<td>Growing table</td>
</tr>
<tr>
<td>Cash_Transaction</td>
<td>CT</td>
<td>8,650,531,346</td>
<td>8,650,531</td>
<td>Growing table</td>
</tr>
<tr>
<td>Trade_History</td>
<td>TH</td>
<td>9,538,576,468</td>
<td>9,538,577</td>
<td>Growing table</td>
</tr>
<tr>
<td>Trade</td>
<td>T</td>
<td>10,222,083,114</td>
<td>10,222,083</td>
<td>Growing table</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>37,634,661,077</strong></td>
<td><strong>705,339,907</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 3-4 indicates the feasibility of partitioning from the viewpoint of table sizes. It displays the initial sizes of all tables in a TPC-E database when the database is populated using default parameters. These highlighted tables are growing tables or scaling/growing tables. Ninety-eight percent of database data are stored in these eight tables. Partitioning
and distributing datasets in these eight tables to multiple shared-nothing database servers will be efficient for both hardware (memory, disk etc) and operation resource (processor time, network bandwidth). In next section we will discuss how to partition these tables using our new partitioning methodology to deliver both high performance and high availability for the whole RAIDB system.

### 3.3.3 Functional Horizontal Partitioning

In previous section, we found out that there maybe more than one table need be partitioned for performance purpose. In order to reduce the message exchange between shared-nothing machines, we want ensure every server be self-reliant as if itself is the only working server. Therefore, each transaction should be contained (handled) by a single server if possible. If a transaction need access a group of tables, all related data should reside on the same server. To fulfill this requirement, we need analyze the relationships among these tables. Figure 3-11 gives the relationships among all entities in the TPC-E database. In Figure 3-11 each relationship between two tables is represented by an arrowed line, where the arrowhead indicates the parent entity. Such references are usually known as foreign keys, which are used to maintain data integrity and enforce data consistency among related tables. If the data in the parent table are partitioned among multiple servers, all tables referencing that table should also be partitioned accordingly. Figure 3-11 shows the table dependency among TPC-E sample database.
In Figure 3-11 all growing tables are highlighted by green color. In this figure, all Growing tables reference table CA (CUSTOMER_ACCOUNT). Table CA references table C (Customer) and B (Broker). Base on the business model and transaction characters, we can choose table C as the root table to partition workload. For simplicity, we will partition these tables by customer ID column, C_ID in Customer table. We use mod function to evenly distribute all customers’ data onto separated partitions. Other partition methods, such as hashing custom’s data by their last four digit tax ID or separating customers in different range by their ID, can also be acceptable for partition purpose. The fundamental requirement is: making each server self-reliant and keep message exchange among servers as little as possible.
Since all partitioning candidates will follow the same partition criteria, related data records in different table will be kept on same partition. We call such partitioning method as Functional Horizontal Partitioning (FHP).

The table dependency displayed in **Figure 3-11** is an ideal case for our partition methodology. All partitioning candidates root from same table, which make it easy to choose partition criteria. If more than one table group need be partitioned, the situation will get more complicated. Suppose there are two table groups need be partitioned, and these two table groups don’t closely related to each other, such as the two highlighted table groups in **Figure 3-12**.

![Figure 3-12 Dependency Tree of TPC-E Database (2)](image-url)
There are three solutions for above situation:

a) Find the base table that both groups root from, such as the Address (AD) table in Figure 3-12. Use Address table as the root table and choose partition criteria from Address table.

b) If there isn’t such mutually shared root table for all table groups, and the queries to access both groups are very infrequent, the whole database can be split into independent databases. Therefore, database administrator can choose different partition criteria for each table group.

c) If there are many join queries frequently access tables in both groups, database administrator may consider to create new tables base these join queries. Partitioning criteria can be derived base on these new tables.

3.4 Data structure and API for PSR system with Customizable partitioning

After selecting the candidate(s) of data partitioning and determine the dependency tree for tables, it is time to upload this information onto the PSR middleware gateway. A data structure and a group of API functions are needed to maintain the partitioning information.

3.4.1 Data Structure

To administrate the partition information over a shared-nothing RAIDB system, the partition information must be maintained by PSR middleware gateway(s). A data structure
(or class) is needed by the PSR middleware gateway. Such data structure should include the following information,

<table>
<thead>
<tr>
<th>Partition Name:</th>
<th>One RAIDB can support one partition schemas for each database. Each partition schema must use a unique schema Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Server Count:</td>
<td>This is the number of all servers in the RAIDB architecture. This is a global variable and it can be maintained at the RAIDB structure level.</td>
</tr>
<tr>
<td>Total Partition Count:</td>
<td>The number of logical partitions created for this partition schema.</td>
</tr>
<tr>
<td>Partitioning Method:</td>
<td>The partitioning criteria method used for partitioning.</td>
</tr>
<tr>
<td>Partitioning function:</td>
<td>The function or list values used by partitioning method to determine query/transaction destination.</td>
</tr>
<tr>
<td>Partitioning Matrix:</td>
<td>The matrix shows the active partition deployment on all servers for the specified partition schema.</td>
</tr>
</tbody>
</table>

All commonly used partition criteria can be adopted for the PSR partition structure. Figure 3-13, Figure 3-14 and Figure 3-15 give three examples of partition configuration. Figure 3-13 shows an example using a trivial hash (mod) function. Figure 3-14 shows a partition schema example using range partitioning. In Figure 3-14, Parenthesis sign “(” represent “greater than” and “less than”. Bracket signs “[” represents “greater than or equal to” and “less than or equal to” respectively. Figure 3-15 gives an example using list values for partition purpose.
[Partition Schema Name]  
TPCE  
[Server Count]  
5  
[Partition Count]  
12  
[ID|012345678901|Full]  
000|111111111111|1  
001|111100000000|0  
002|000011110000|0  
003|000000001111|0  
004|111111111111|1  
[Partition method]  
0  
[Function]  
%

Figure 3-13 Example of FHP Configuration (1)

[Partition Schema Name]  
T2  
[Server Count]  
4  
[Partition Count]  
6  
[ID|012345|Full]  
000|111111|1  
001|110000|0  
002|001100|0  
003|000011|0  
[Partition method]  
1  
[Range values, bottom<x<=cell]  
(0000:1000]  
(1000:5000]  
(5000:99999]

Figure 3-14 Example of FHP Configuration (2)

[Partition Schema Name]  
T3  
[Server Count]  
4  
[Partition Count]  
4  
[ID|0123|Full]  
000|1111|1  
001|1000|0  
002|0100|0

51
In above partition configuration examples, the partition deployment matrix includes three parts: the server ID, the active partitions on that server and the server type bit. The server type bit can help gateway quickly pick out those full servers. It is also useful to help database administrator quickly mark all partitions on full servers. Maintaining the server type bit also saves processing time by skip the evaluation of each server for every incoming client query.

### 3.4.2 API functionalities

To administrate the partition configuration on PSR middleware gateway, the following API functions are needed to provide an interface between user and PRS gateway:

```c
int GetPartitionCount()

struct GetServerPartition(int ServerIdx)

int SetServerPartition(struct PartitionStr)

int FindNextFullServer()

int FindSingleSlimServer(int nPartIdx)

int FindSlimServers(int nIdx, int nPartIdx)
```
3.5 Summary

This chapter describes the fundamental design of the new Functional Horizontal Partitioning (FHP) methodology for a RAIDB system using parallel synchronous transaction replication technology. The execution frequency and footprints of transactions need be analyzed to choose partitioning candidates. Table dependencies among all database relationships are used to put all candidate entities into one group. Partitioning criteria can be determined based on the root table in this group. This Functional Horizontal Partitioning keeps each member servers self-reliant and most transactions can be processed by single servers. It also minimizes the message exchange among servers. RAIDB system using Functional Horizontal Partitioning can deliver both high performance and high availability. The data structure and API function needed by this methodology are also discussed.
CHAPTER 4 PRML FOR DATABASE WITH VARIABLE REDUNDANCY

Let’s go back to the call center example in Chapter 3 for a quick fact check. An automated menu is used by the call center to forward customers’ calls to corresponding agent(s). This middle-way menu is critical for the whole system. Without this automated menu, a customer may end up to an agent who knows nothing about the customer’s concern, or the manager will have to exclusively hire senior agents to do many junior agents’ work, which will be very inefficient and costly. DBx gateway can serve as such ‘automated menu’ by parsing customer’s request. It also needs clients’ input to narrow down their options (choices). How can we implement that functionality with minimal re-coding efforts? The answer is extending the Replication Markup Language for parallel synchronous transaction replication architecture integrated with the new Functional Horizontal Partitioning technology.

This chapter presents the specification of the Partition Replication Markup Language in detail. We will first go over all PRML tags supported by this language, and then we will use a sample program to show how to use the PRML for PSR system integrated with Functional Horizontal Partitioning technology.
4.1 PRML Tag for Functional Horizontal Partitioning

Four new PRML tags are designed for the usage of Functional Horizontal Partitioning schema. These extended PRML tags include the following new members:

- Single Partition tags (*READSLIM* tag and *WRITESLIM* tag)
- Cross partition scan tag (*READFULL* tag)
- Replication control tag (*WRITETOALL* tag)

These tags help to

- Provide information to identify the target partition
- Maintain object lock
- Balance read queries among proper servers
- Broadcast write actions to proper servers (within specified partition or across all servers)

Like old RML tags, the new PRML tags include

- Specified header
- A token tag
- Token switch
- Object ID and lock level (for *READSLIM* tag and *WRITESLIM* tag only)

Not only these new PRML tags can be manually inserted into application code by programmer, these tags can also be automatically inserted on session layer for each
connection. For example, suppose we have a database with transaction record table partitioned by customer ID. After a customer login into this database, a PRML partition tag with customer’s account information can be inserted before the customer’s first query, the customer’s query will be automatically forwarded to the server with the target partition. Another PRML partition closing tag can be inserted before the customer disconnect from database. This feature can keep the code change to the minimal level.

Another advantage of these new PRML tags is that application code using PRML statements can work properly on a single full server even without the RAIDB gateway in the map. It guarantees a safe fallback path for DBA and Database developer. They can develop their code on a single full server node and deploy their code to a RAIDB system. They can safely get back to their original single server system, although unlikely, if they don’t want the RAIDB architecture and they don’t need to worry that their code may behave differently.

### 4.1.1 Single-Partition Tags

A single-partition tag identifies the action to be processed on specified partition. Instead of broadcasting to every server member in the cluster, RAIDB middleware will forward the queries wrapped by single-partition tags to the servers where the specified partition resides. Single-partition tag cannot be nested to itself. There are two single-partition tags: READSLIM and WRITESLIM.
READSLIM tags are used to wrap read-only queries. These queries will be load balanced among proper servers. READSLIM tags provide the following functions.

- Provide partition key
- Load balance read-only queries on specified partition
- Maintain specified object in shared or exclusive mode

A READSLIM tag is used in the format of “--SET ICXP PartitionName READSLIM ON/OFF Partition_Key lock_mode”.

Example 1

1. `select CA_C_ID from dbo.CUSTOMER_ACCOUNT where CA_ID = 43000006771`
2. `--SET ICXP TPCE READSLIM ON 4300000678 0`
3. `exec dbo.spr_Trade_Status 43000006771`
4. `--SET ICXP TPCE READSLIM OFF 4300000678`

Figure 4-1 Example of READSLIM Tag

In Example 1, line 1 is used to obtain the partition key (4300000678) for customer_account (43000006771) from database server, line 2 and 4 use READSLIM tags to wrap around the read-only stored procedure in line 3. A shared-mode (0) lock is obtained for object 43000006771. The stored procedure will be executed on proper server to obtain the correct result set. The shared mode lock is released at line 4. The lock mode is not needed when it is released.
WRITESLIM tags are used to wrap write operations on specified partition. These operations will be broadcasted to all servers hosting the specified partition. WRITESLIM tags provide the following functions.

- Provide partition key
- Broadcast write queries to target servers with specified partition
- Maintain specified object in exclusive-mode

WRITESLIM tags are used in the format of “--SET ICXP PartitionName WRITESLIM ON/OFF Partition_Key”. Figure 4-2 and Figure 4-3 show two examples of WRITESLIM tags.

- Example 2

1. select TC_C_ID from dbo.TRADE_CID where TC_T_ID = 20000000230570
2. --SET ICXP TPCE WRITESLIM ON 4300000560
3. Exec dbo.spw_Trade_Update_F1b 200000000230570, 'Kelly X Wehr'
4. --SET ICXP TPCE WRITESLIM OFF 4300000560

Figure 4-2 Example of WRITESLIM Tag

-Example 3

1. select TC_C_ID from dbo.TRADE_CID where TC_T_ID = 200000002562815
2. --SET ICXP TPCE READSLIM ON 4300000730 1
3. Exec dbo.spr_Trade_Update_F1a 200000002562815
4. --SET ICXP TPCE WRITESLIM ON 4300000730
5. Exec dbo.spw_Trade_Update 200000002562815,'Roberto X Palfreyman'
6. --SET ICXP TPCE WRITESLIM OFF 4300000730
7. Exec dbo.spr_Trade_Update_F1c 200000002562815
8. --SET ICXP TPCE READSLIM OFF 4300000730

Figure 4-3 Example of Nested Partition Tags
In Example 3, one pair of WRITESLIM tags are wrapped between a pair of READSLIM tags. The partition key (4300000730) used by WRITESLIM and READSLIM tags is obtained by line 1. The READSLIM tags provide two functionalities in this example, 1) it indicates the wrapped statements can be load balanced unless other PRML tags take effect; such nesting structure keeps the load of read and write to minimal level; 2) it obtains a exclusive lock on object (4300000730) and keep other connections from accessing this object. Only WRITESLIM tags embedded in the same connection can safely access the object. The WRITESLIM tag at line 4 will automatically change the lock to exclusive level if that lock is in shared level. Because of the nesting feature, a connection doesn’t need to release a lock before obtaining exclusive lock on the same object. The WRITESLIM tag at line 6 closes the WRITESLIM control. It will release the lock control back to the READSLIM control.

4.1.2 Cross Partition Scan Tag (READFULL tag)

READFULL tags are used when a query needs to scan the whole virtual table. The READFULL tag is designed for this purpose.

The READFULL tag provides the following functions

- Specify the partition range (all partitions) where the wrapped query to be executed
- Load balance the wrapped query among proper servers
READFULL tags are used in the format of “--SET ICXP PartitionName READFULL ON/OFF”.

- Example 4

1    --SET ICXP TPCE READFULL ON
2    Exec dbo.spr_MarketFeed_2 'ZONSPRC', 29.53, 'TSL', 'TLS', 'TLB'
3    --SET ICXP TPCE READFULL OFF

Figure 4-4 Example of READFULL Tag

READFULL tag is usually used for Decision Support queries. It has the highest privilege among all PRML tags except ICXNR tag. Other PRML tags wrapped between READFULL tags will be ignored. We will discuss the privilege rank of PRML tags in Chapter 5.

4.1.3 Replication control tag (WRITETOALL tag)

Under some scenario, one connection working on a specified partition needs to update some non-partition tables. To keep data consistency, such write statements must be broadcasted to every server in the RAIDB architecture. WRITETOALL tag is designed to help programmer to broadcast those specified write statements onto non-partitioned tables (resides on every member server) without leaving current partition working section. WRITETOALL tags provide the following functions

- Broadcast the wrapped statement to every server
- Maintain obtained object lock and partition configuration for current connection
WRITETOALL tags are used in the format of “--SET ICXP PartitionName WRITETOALL ON/OFF”.

-Example 5

1  --SET ICXP TPCE READSLIM ON 4300000048
2    Exec dbo.spr_Trade_Order_F3d 43000000471, 'AXSI'
3  --SET ICXP TPCE WRITETOALL ON
4    Exec dbo.spw_Ins_TID 200000002909822, 4300000048
5  --SET ICXP TPCE WRITETOALL OFF
6  --SET ICXP TPCE READSLIM OFF 4300000048

Figure 4-5 Example of WRITETOALL Tag

The WRITETOALL tags should be used within the scope of READSLIM or WRITESLIM tags. It is not harmful to use WRITETOALL tags outside of READSLIM or WRITESLIM tags, but it is useless because RAIDB middleware will automatically broadcast such write statements to every database server.
4.2 An Example of PRML

Figure 4-6 gives an example using new PRML tags we discussed in previous section.

```sql
1  select C_ID from dbo.CUSTOMER where C_TAX_ID = '802WA7622ME750'
2  --SET ICXP TPCE READSLIM ON 4300000048
3  Exec dbo.spr_Trade_Order_F3d 43000000471, 'AXSI'
4  begin tran
5  --SET ICXNR ON
6     Exec dbo.spw_Trade_Order_F4_GetNewTID 4300000048
7  --SET ICXNR OFF
8  --SET ICXP TPCE WRITETOALL ON
9     Exec dbo.spw_Ins_TID 200000002909822, 4300000048
10  --SET ICXP TPCE WRITETOALL OFF
11  --SET ICXP TPCE WRITESLIM ON 4300000048
12     Exec dbo.spw_Trade_Order_F4 200000002909822,
13  commit
14  --SET ICXP TPCE WRITESLIM OFF 4300000048
15  --SET ICXP TPCE READSLIM OFF 4300000048
```

**Figure 4-6 Example of PRML Tags**

The SELECT statement in line 1 retrieves the partition key (4300000048) for the rest of the code. In the nesting structure of PRML statements, a transaction is started at line 4 on the specified object. Line 5-7 is used to obtain a unique new trade ID, which is generated by a non-deterministic function at server side. Line 8-10 is used to broadcast this new ID to all servers in this RAIDB system. The new ID will be ignored by the server which generated it
(we will discuss the use of non-deterministic functions in RAIDB system in Chapter 6). In line 11-14, the transaction updates the partition object (4300000048) and commits the change. There is one thing need here need our attentions. The “begin tran” statement at line 4 is outside of the WRITESLIM tags, while the “commit” statement is in the scope of WRITESLIM tags between line 11 and 14. Actually, it doesn’t matter where the “begin tran” and “commit” are wrapped, because RAIDB gateway will broadcast such transaction control statements to every member server. We will discuss the new PRML parser in Chapter 5.

4.3 Summary

This chapter presents the new tags of Partition Replication Markup Language (PRML) designed for parallel synchronous transaction replication system using Functional Horizontal Partitioning schema in a shared-nothing database cluster.

To the best knowledge of the author, the Partition Replication Markup Language and Functional Horizontal Partitioning technology are the first attempts of solving database replication on a self-resilient database cluster with customizable data redundancy schema. Compared with other approaches, PRML is readable, portable and fully compatible with any existing database application.
CHAPTER 5 PARSEING PRML TAGS

The PRML parser contains three major phases, PRML tag parsing, identifying target server(s) and processing query to/from target server(s), as shown in Figure 5-1.

For each connection, the PRML Parser keeps a Tag Mask through the lifetime of the connection. The PRML Parser extracts PRML tags from client queries and updates the Tag Mask of that connection. Using the information in that Tag Mask, PRML Parser can identify the target server(s) for each query, and the target server(s) information will be
stored in the Server Mask. Finally, the client query is forwarded to its target server(s), and the result from target server(s) will be returned to client. This chapter will illustrate each of these phases in detail.

5.1 The PRML Tag Priority and PRML Mask

The core of PRML parsing is a PRML mask. For each logic connection between client and RAIDB, the RAIDB maintains a PRML mask throughout the lifetime of that connection. The PRML mask can be declared as an integer. Each tag is represented by one bit.

<table>
<thead>
<tr>
<th>Tag Type</th>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrimaryOnly</td>
<td>Bit</td>
</tr>
<tr>
<td>ReadFull</td>
<td>Bit</td>
</tr>
<tr>
<td>ReadSlim</td>
<td>Bit</td>
</tr>
<tr>
<td>WriteSlim</td>
<td>Bit</td>
</tr>
<tr>
<td>LoadBalance</td>
<td>Bit</td>
</tr>
<tr>
<td>WriteToAll</td>
<td>Bit</td>
</tr>
<tr>
<td>Read</td>
<td>Bit</td>
</tr>
</tbody>
</table>

All PRML tags from client packets are filtered and validated against the definition of PRML. These tags will trigger the update of PRML mask states. These states in PRML mask act like switches on a control board. These “switches” decide which server(s) will be the target servers for incoming client queries. The PRML mask maintains all the status of all PRML tags. The status of each tag has different priority. The order of these priorities from higher to lower is following

$$ICXNR > READFULL > WRITEALL > READSLIM = WRITESLIM > ICXLB$$  \hspace{1cm} \text{Eq. 5-1}

When the state for a higher priority tag is ON, it will affect the path of each client query goes. Further more, the states of some lower priority tags maybe overshadowed by higher priority tags.
The ICXLOCK tag is not included in the priority chain in Eq. 5-1. It is because that ICXLOCK tag is used to serialize concurrent accesses on specific object(s). ICXLOCK tag does not affect the path a query goes through.

5.2 The PRML Parser and Flow Control

The PRML parser uses the status of the PRML mask to determine the path for each incoming client request. This parser is an extension of the original RML Parser shown in Figure 2-5. The new PRML parser is shown in Figure 5-2.

![Figure 5-2 PRML Flow Control](image-url)
In Figure 5-2, the PRML parser uses the information maintained in the PRML Mask (not shown in Figure 5-2) to determine how to handle incoming client queries. There are seven terminators (in oval shape) in Figure 5-2. Once the parser ends in one of the seven terminators, the content of Server Mask is settled. It will be used to forward incoming regular queries later.

PRML statements must be submitted separately because those PRML tag packet will be filtered by RAIDB gateway and won’t be forwarded to database server. Regular client query packets will not be parsed but their destinations are determined by the state of the PRML Mask.

5.3 Lock Isolation Control by PRML statement

Lock control is needed by all relational database applications. From the isolation level, it usually includes table lock, page lock and row lock. Page lock is closely related to the hardware and operating system on the host machine. It is maintained by database engine and is out of the access of a third party gateway. On the other hand, table lock and row lock are two logic locks that can be supported by our PSR gateway.

PSR gateway sets exclusive table locks on database objects when a client modifies those entities. Such table locks save operating resource, but they sacrifice the concurrency and overall system performance. The ICXLOCK tag can support high concurrency by providing the finest-grained row level lock.
However, row level lock maybe too fine to be used in complicated transaction when multiple tables will be modified within the same transaction. For example, one transaction executed by a travel agent may need to update or insert records in different tables. Figure 5-3 shows the SQL statements to be executed on a database server without PSR gateway. Three tables will be modified in this transaction.

```
begin tran

--pull out customer id for an existing customer by email account
exec spr_FindCustomerID 'fxiong@temple.edu'

--update customer's credit card, passport information
update Customer
set CCID='7789-2315-5671-1998',
     PPID='G179805698'
where email='fxiong@temple.edu'

--use customer's credit card to reserve hotel
insert into HotelReservation(CreditCard, CheckInDate, HotelID)
    values('7789-2315-5671-1998', '04-28-2009', 'H1005')

--user customer's passport number for airline ticket reservation
insert into TicketReservation(PPID, AirlineID, FlightNum, DepartDate)
    values('G179805698', 'AA', 'G87', '04-28-2009')

commit
```

**Figure 5-3 Example of SQL Statements without PSR Lock Control**

In order to put row level lock control via PSR gateway, PRML tags can be added into the code in Figure 5-3. If we only use ICXLOCK tag, the rewritten code will be look like the one in Figure 5-4. Three locks will be needed for three separated tables.
begin tran

--pull out customer id for an existing customer by email account
exec spr_FindCustomerID 'fxiong@temple.edu'

--SET ICXLOCK ON 'fxiong@temple.edu'
update Customer
set CCID='7789-2315-5671-1998',
PPID='G179805698'
where email='fxiong@temple.edu'

--SET ICXLOCK ON '7789-2315-5671-1998'
insert into HotelReservation (CreditCard, CheckInDate, HotelID)
values('7789-2315-5671-1998', '04-28-2009', 'H1005')

--SET ICXLOCK ON 'G179805698'
insert into TicketReservation(PPID, AirlineID, FlightNum, DepartDate)
values('G179805698', 'AA', 'G87', '04-28-2009')

commit

--SET ICXLOCK OFF 'G179805698'
--SET ICXLOCK OFF '7789-2315-5671-1998'
--SET ICXLOCK OFF 'fxiong@temple.edu'

Figure 5-4 Example of Lock Control using RML Tags

Maintaining locks is an expensive operation for all software applications. Above example shows the limitation of ICXLOCK tag. Multiple locks will be held for a single transaction in a single connection. It will cost a lot system resource on the PSR gateway host machine, especially when the connection pool gets bigger and bigger. PRML tags “READSLIM” and “WRITESLIM” can be used to relieve such lock cost. Suppose all three tables in Figure 5-3 and Figure 5-4 can be partitioned using our new FHP method presented in Chapter 3. We can partition these tables using customer ID (email). The code in Figure 5-4 can be rewritten as Figure 5-5. In this rewritten code, we only need one partition lock for the whole transaction. Therefore, the new PRML tags for FHP can save a lot of resource and make programmers’ work easier to be migrated.
begin tran

--SET ICXLB ON
--pull out customer id for an existing customer by email account
exec spr_FindCustomerID 'fxiong@temple.edu'
--SET ICXLB OFF

--SET ICXP TPCE WRITESLIM ON 'fxiong@temple.edu'
update Customer
set CCID='7789-2315-5671-1998',
          PPID='G179805698'
where email='fxiong@temple.edu'

insert into HotelReservation (CreditCard, CheckInDate, HotelID)
values('7789-2315-5671-1998', '04-28-2009', 'H1005')

insert into TicketReservation(PPID, AirlineID, FlightNum, DepartDate)
values('G179805698', 'AA', 'G87', '04-28-2009')

commit
--SET ICXP TPCE WRITESLIM OFF 'fxiong@temple.edu'

Figure 5-5 Example of Lock Control using PRML Tag

5.4 Summary

The PRML parser filters PRML tags from client packets and maintains a PRML Mask for each connection. The destination server(s) for each client query is determined by the state of corresponding PRML Mask. There is a priority rank among all PRML tags and higher rank tag may overshadow lower rank tags. The PRML parser doesn’t parse regular communication packets; theoretically the proposed PRML parser can be used to process any communication between clients and database servers.
The PRML tags and PRML Parser provide a new layer of object lock control, partition object lock, which exists between coarser-grained table lock and the finest-grained row lock. Using partition object lock can save a lot of operating resource for the whole RAIDB system.
CHAPTER 6 TRANSACTION CONTROL IN PSR SYSTEM

With multiple synchronized datasets maintained in a RAIDB system, the administration work gets more complicated than a system with single date copy. This chapter presents a few problems incurred by RAIDB technology and our solutions to handle these problems in a RAIDB system.

6.1 Transaction Consistency Control

In an OLTP system without redundant data copy, any transaction can have only two statuses: success or fail. In a database cluster with multiple data replicas, a submitted transaction may be successful on one server while be unsuccessful on another node. For a system using Two Phase Commit protocol, such transaction must be rolled back. Original DB3 system uses the result from a primary server, which can be manually assigned or automatically selected, as the correct response about whether the submitted transaction has been committed or not. Servers returning response different from the primary server will be declared as unreliable server and removed from the cluster. This policy guarantees data consistency among all active servers, but it may deactivate a healthy server simply by some unexpected error, such as short communication interruption or delayed response. In a busy system, deactivating a working server means more load to be distributed to other working servers. It will keep these servers busier and more delay may happen, thus more server may
be deactivated one by one; the whole system performance and availability will get deteriorate.

To solve this problem, a new transaction commit policy combining the features of 2PC and DBx is developed. A frequency of transaction inconsistency is predefined for each server in a RAIDB system. If the number of transaction inconstancy happened in a unit time is less than the predefined value, RAIDB gateway will rollback that transaction and give the corresponding server a second chance. Client may choose to abort the transaction or resubmit that transaction. (We will discuss automatic transaction resubmission in next section). Once the transaction inconstancy frequency reaches the predefined threshold, the troubling server will be dismissed from the cluster. Therefore, the throughput of the whole system can be improved by removing unreliable server; and the performance and availability won’t be affected by some contingent events.

6.2 Transaction Resubmission during Gateway Failover

In a regular OLTP system without parallel synchronous transaction replication, the transaction submission is very simple, shown in Figure 6-1. After a connection is built and the SQL statement is prepared, a try/catch mechanism can wrap the statements to be executed on server side. If an error is detected by those database API functions or database drivers, the error will be caught and handled. After that the SQL statement can be resubmitted to database server again. The transaction status is accurate at any moment during the whole procedure.
For an OLTP system using parallel synchronous transaction replication, the error-handling logic in Figure 6-1 is not enough. For example, one transaction is submitted by client and accepted by the RAIDB gateway. If the RAIDB gateway crashes while it is committing the transaction on every database servers, the submitted transaction may get committed on some server but not committed on other server. The transaction status among database servers will be inconsistent, which cannot be handled by the gateway because it just crashed.
6.2.1 Transaction Resubmission without Connection Pool

To avoid the transaction inconsistency, the error-handling procedure in Figure 6-1 needs be adjusted for RAIDB system, shown in Figure 6-2. Here are the major changes needed be included:

![Simple Synchronous Code with Fail Over Recovery](image)

- Initialize and collect data from client side
- Build SQL connection
- Build SQL Command (with parameters)
- Submit SQL command
- Error?
  - Yes: Rebuild Connection
  - No: Close connection
- Connection lost?
  - Yes: Use other error handler
  - No: Other error handler code
- Retry?
  - Yes: Rebuild Connection
  - No: Report result and exit

**Figure 6-2 Transaction Resubmission without Connection Pool**

a) Assign a standby RAIDB gateway (not shown in Figure 6-2) to monitor the status of working RAIDB gateway. It will take over all service once it detects the crash at working RAIDB gateway. All existing connections through the crashed gateway will be disconnected, and new connection will go through the new acting gateway.
b) Unique transaction ID must be used for each submitted transaction (which is a reasonable requirement on a normalized database). Such unique transaction ID can be obtained from client side or generated at server side. Such transaction ID will be used later to determine the existence of a transaction exists on a database server.

c) Create stored procedures on each database server to handle the transaction submitted by clients. It will skip a committed transaction and only commit new transaction. The execution result will be returned to client as server response.

d) Once a lost-connection error is detected, client will rebuild the connection and resubmit the transaction to RAIDB system. The resubmitted transaction will be handled by the stored procedure in step c. No matter what the final response from each server, we can be sure that the transaction status on all active servers are consistent.

6.2.2 Transaction Resubmission with Connection Pool

Building database connections costs a lot of system resource. In order to provide higher performance, many software vendors use connection pool to maintain a cache of database connections so that a connection can be reused for new incoming client request. In such scenario, the transaction resubmission procedure in Figure 6-2 needs be adjusted as Figure 6-3. In Figure 6-3 individual client process is released from the burden of connection maintenance. A separated manager thread is created to maintain the connection pool. The requirements for standby RAIDB gateway and unique transaction ID are the same.
6.3 Use Non-Deterministic Functions in RAIDB System

Database vendors provide many useful non-deterministic functions for customers. Non-deterministic functions are functions that may return different results each time they are called with a specified input. Functions such as Getdate(), NEWID(), RAND() are typical non-deterministic function. When the return values of these functions are used directly at server side, they may cause data inconsistency among database servers in RAIDB architectures. Programmer should avoid such non-deterministic function or use the PRML statement we discussed in last chapter to keep data consistency among all servers. In this section we will use two examples to show how to maintain data consistency in RAIDB system when calling non-deterministic functions.
Example 1 – Use GetDate() function

Table used in Example 1

```sql
Create table myTable1(
    col1 int not null,
    col2 Datetime not null
)
```

Figure 6-4 Table Definition for Figure 6-6

Client code for system without parallel synchronous replication

```sql
1 Void InsertMyTable(int ID)
2     Insert into myTable (col1, col2) values(ID, GetDate())
3     Return
```

Figure 6-5 Example of Function Called by Client (1)

Client code for system using parallel synchronous replication

```sql
1 InsertMyTable(int ID)
2     Declare myDate Datetime
3 --SET ICXLB ON
4     Select myDate = GetDate()
5 --SET ICXLB OFF
6     Insert into myTable (col1, col2) values(ID, @myDate)
7     Return
```

Figure 6-6 Example of Replication Control (1)

In Figure 6-6, tag ICXLB tags are used in line 3 and line 5. The value of myDate at line 4 will be obtained from a single server in RAIDB system. When client program executes the insert statement at line 6, each server will receive the same value for col2.
Example 2 – Obtain a new ID from database

Table used in Example 2

```sql
Create table myTable2 (  
    col1 uniqueidentifier not null DEFAULT newid(),  
    col2 int          not null  
)
```

Figure 6-7 Table Definition for Figure 6-9

Client code on system without parallel synchronous replication

```sql
1  InsertMyTable2(int salary)
2       Insert into myTable2 (col2) values(salary)
3       Return
```

Figure 6-8 Example of Function Called by Client (2)

Client code on system using parallel synchronous replication

```sql
1  InsertMyTable2(int Salary)
2       Declare NewId uniqueidentifier  
3             --SET ICXNR ON  
4        Select NewId = newid()  
5             --SET ICXNR OFF  
6       Insert into myTable2 (col1, col2) values(NewId, Salary)
7       Return
```

Figure 6-9 Example of Replication Control (2)

Tag ICXNR is used in Figure 6-9. This tag directs client query to the primary server in RAIDB system. In this example, the value of NewID is obtained from primary server and be broadcasted to every server later.
From above two examples, it shows that data inconsistency among RAIDB servers can be avoided when using non-deterministic functions. The returned values of non-deterministic functions should be retrieved to client side and then broadcast to every server.

6.4 Summary

This chapter discusses a few issues about transaction control and data consistency related to parallel synchronous transaction replication technology. The solutions we present in this chapter can help to deliver a seamless non-stop service for RAIDB architecture.
Chapter 7 PERFORMANCE MODELING AND COMPUTATION RESULTS

With the new FHP technology integrated into RAIDB system, not only read-only queries can be load balanced, write operations on partitioned objects can be distributed too. This chapter will discuss the performance modeling of RAIDB using FHP technology.

7.1 Throughput Speedup of RAIDB system

The core of RAIDB system is the middleware gateway. It intercepts clients’ requests and distributes these requests among shared-nothing servers. With the new FHP technology and PRML parser, we need to analyze the throughput speedup model for a RAIDB system, and answer the following critical questions:

a) What is the maximal throughput a RAIDB system can achieve?

b) What is the marginal contribution of each new server node can bring?

c) What is the optimal partition number for a RAIDB system?

The overall load on a database system is composed by Read load and Write load, the latter can be decomposed as Write on non-partitioned objects and Write on partitioned objects, see Eq. 7-1. $L_{base}$ is the base load submitted from client side.

\[ L_{base} = L_{read} + L_{write} = L_{read} + (L_{wp} + L_{wnp}), \]  

Eq. 7-1

$L_{wp}$ is the write load on partitioned objects,

$L_{wnp}$ is the write load on non-partitioned objects
In a RAIDB system using FHP design, the write load will be replicated. For each write operation on non-partitioned object, the gateway will replicate N operation to N active servers. For each write operation on partitioned object, the gateway will replicate R operations to R replicas of the partitioned object. The total load on the RAIDB system is shown in Eq. 7-2.

\[ L_{RAIDB} = L_{\text{read}} + L_{wp} \times R + L_{wp} \times N \]  

\[ N = \text{number of servers, } R = \text{Redundancy degree, replica number of partitioned object} \]

For simplicity, we suppose that each operation consumes the same amount of resource (processor time, memory space, and physical disk access time etc.) no matter it is read-only query or a write operation. We also suppose the load can be evenly distributed among all server nodes. We can get the average load on each server and the total system throughput from Eq. 7-3, Eq. 7-4 and Eq. 7-5.

\[ L_{\text{average}} = (L_{\text{read}} + L_{wp} \times R + L_{wp} \times N) / N \]  

Eq. 7-3

\[ \text{Throughput}_{\text{Base}} = W / L_{\text{Base}}, \text{ } W \text{ is the performance power of single server in unit time} \]

Eq. 7-4

\[ \text{Throughput}_{\text{RAIDB}} = (N \times W) / (L_{\text{read}} + L_{wp} \times R + L_{wp} \times N) = W / L_{\text{average}} \]

Eq. 7-5

Eq. 7-6 shows the throughput speedup of a RAIDB system. Its upper limit is determined by the write load on non-partition objects, see Eq. 7-7.

\[ S_p(N, R, L_{\text{read}}, L_{wp}) = \frac{\text{Throughput}_{\text{RAIDB}}}{\text{Throughput}_{\text{Base}}} = \frac{W / L_{\text{average}}}{W / L_{\text{Base}}} = \frac{L_{wp} \times L_{wp} \times L_{\text{read}}}{L_{wp} + (L_{wp} \times R + L_{\text{read})} / N} \]

Eq. 7-7
\[
\lim_{N \to \infty} S_p = \frac{L_{\text{wnp}} + L_{\text{wp}} + L_{\text{read}}}{L_{\text{wnp}}}
\]

Eq. 7-7

Since the degree of redundancy, \(L_{\text{read}}\) and \(L_{\text{wnp}}\) are usually predetermined, the throughput speedup is a function of server number \(N\). We can get the marginal contribution of adding a new server into a RAIDB, as shown in Eq. 7-8.

\[
S_p(N) = \left(\frac{L_{\text{wnp}} + L_{\text{wp}} + L_{\text{read}}}{L_{\text{wnp}} + (L_{\text{wp}} \times R + L_{\text{read}}) / N}\right) = \frac{L_{\text{wp}} \times R + L_{\text{read}}}{(L_{\text{wnp}} \times N + L_{\text{wp}} \times R + L_{\text{read}})^2}
\]

Eq. 7-8

Eq. 7-7 and Eq. 7-8 are simple estimates of the throughput enhancements of a RAIDB system, since all queries are assumed be homogeneous. However, they do reveal the existence of upper limit of a throughput speedup and the diminishing of marginal contribution. Besides the effect of \(N\), the redundancy degree also plays an important role. With greater redundancy, the marginal contribution of throughput speedup in Eq. 7-8 drops faster.

Figure 7-1 shows the calculated theoretical throughput speedup of a RAIDB system can achieve when it maintains two synchronous copies of datasets (there are \(N\) copies of non-partitioned dataset in the system, of course.) The total base load is 1. The ratio of read-only queries is 0.8, and the total ratios for write operations is 0.2, which includes write operations on partitioned objects (Wp) and write operations on non-partitioned objects (Wnp). This figure lists three different combinations of Wnp and Wp, and one additional base case of a regular PSR system with full redundancy on each node (the blue curve). As
we can see, the maximal throughput speedup for a PSR system with full data redundancy is barely 5 (when the ratio of read-only queries is 80%).

Since the marginal contribution of throughput speedup keeps on decreasing, it is not cost-efficient to pursue the theoretical maximal throughput speedup. Here we introduce two terms, \(N_{0.5}\) and \(N_{0.25}\), which respectively represent the number of minimal necessary server nodes to achieve half of the theoretical maximal throughput speedup and a quarter of the theoretical maximal throughput speedup. The calculation procedure for \(N_{0.5}\) is displayed in Eq. 7-9 and Eq. 7-10.
\[ Sp(N_{0.5}) = \frac{L_{\text{wp}} + L_{\text{wp}} + L_{\text{read}}}{L_{\text{wp}} + (L_{\text{wp}} \times R + L_{\text{read}}) / N_{0.5}} = \frac{1}{2} Sp_{\text{max}} = \frac{1}{2} \frac{L_{\text{wp}} + L_{\text{wp}} + L_{\text{read}}}{L_{\text{wp}}} \]

Eq. 7-9

\[ N_{0.5} = \frac{L_{\text{wp}} \times R + L_{\text{read}}}{L_{\text{wp}}} \]

Eq. 7-10

Similarly, we can get the value of \( N_{0.25} \) for a quarter of the theoretical maximal throughput scaleup in Eq. 7-11.

\[ N_{0.25} = \frac{L_{\text{wp}} \times R + L_{\text{read}}}{3L_{\text{wp}}} \]

Eq. 7-11

Using Eq. 7-6, Eq. 7-10 and Eq. 7-11, user can estimate the potential throughput speedup and the number of nodes needed for a RAIDB system. Table 7-1 gives a few examples of \( N_{0.5} \) and \( N_{0.25} \), along with their corresponding throughput speedup values.

<table>
<thead>
<tr>
<th>Redundancy of Partitioned Data</th>
<th>2</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>80%</td>
<td>80%</td>
<td>80%</td>
</tr>
<tr>
<td>Wnp</td>
<td>10%</td>
<td>1%</td>
<td>0.1%</td>
</tr>
<tr>
<td>Wp</td>
<td>10%</td>
<td>19%</td>
<td>19.9%</td>
</tr>
<tr>
<td>Upper limit of Sp</td>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>( N_{0.5} ) and ( Sp(N_{0.5}) )</td>
<td>10 (5)</td>
<td>118 (50)</td>
<td>1198 (500)</td>
</tr>
<tr>
<td>( N_{0.25} ) and ( Sp(N_{0.25}) )</td>
<td>4 (2.857)</td>
<td>40 (25.32)</td>
<td>400 (250.3)</td>
</tr>
</tbody>
</table>

Table 7-1 Examples of \( N_{0.5} \) and \( N_{0.25} \)
7.2 Timing Models for RAIDB System

The discussion in previous section reveals the trend of throughput speed in a RAIDB system. It is based on the assumption that all client requests are similar and the cost to process each request is the same. This section we analyze the timing model for select, insert, delete and update request in a RAIDB system.

7.2.1 Timing Model for Insert, Update and Delete Operations

In a normalized database, we can safely assume that each table maintains a primary key or unique index for each tuple. Such primary key and index are usually stored in balanced search tree structures, such as AVL balanced trees or Red-Black trees.

As we know, all basic operations take $O(\log n)$ time on a balanced tree structure. For a bunch of write operations including $m$ write requests (update, insert or delete) on a size-$n$ table, the sequential timing model is

$$T_{seq} = \frac{m \times \log(n)}{W}, \quad W \text{ is the power of single server} \quad \text{Eq. 7-12}$$

If all these $m$ write requests are on partitioned objects and can be evenly distributed on all member servers in a RAIDB system whose server number equals $N$ and data redundancy equals $R$, the corresponding parallel timing model is shown in Eq. 7-13.
Therefore, the throughput speedup for this bunch of write requests on partitioned objects can be shown in Eq. 7-14

$$Sp = \frac{T_{seq}}{T_{para,p}} = \frac{\frac{N \times lg(n)}{R \times [lg(n) - lg(p)]}}{\frac{R \times m \times lg(n/p)}{N \times W}} = \frac{N}{R} \lim_{n \to \infty, n \gg p} \quad \text{Eq. 7-14}$$

Above throughput speedup equation indicates that there won’t be any speedup if every server keeps a full redundant copy. It proves that partitioning data among servers is mandatory for throughput speedup on write operations.

### 7.2.2 Timing Model for Select Operations

If there are m select queries to be executed and each query only needs to retrieve one row from one table, furthermore, if all these select queries can be evenly distributed, the throughput speedup will be close to linear speedup, as shown in Eq. 7-15.

$$Sp = \frac{T_{seq}}{T_{para,p}} = \frac{m \times lg(n) / W}{m \times [c + lg(n/p)]/(NW)} \rightarrow \lim_{N \to \infty, N \gg c} \quad \text{Eq. 7-15}$$

$c$ is the overhead of partition determination, it is usually a constant.
The timing model for a SELECT query involving with \( k \) tables is a little more complicated, especially when multiple rows to be retrieved. Suppose the average number of rows in each of these \( k \) table, \( T_1, T_2, \ldots, T_k \), is \( n \), we can get the sequential timing model for a SELECT query as following,

\[
T_{\text{seq}} = \frac{n^k}{W} \tag{Eq. 7-16}
\]

If each of these tables is evenly divided into \( p \) partitions, and all partitions reside on one server, as shown in Eq. 7-17.

\[
\left\{ \begin{array}{ccc}
T_{11} & T_{12} & \cdots & T_{k1} \\
T_{12} & T_{22} & \cdots & T_{k2} \\
\vdots & \vdots & \ddots & \vdots \\
T_{1p} & T_{2p} & \cdots & T_{kp}
\end{array} \right\} \tag{Eq. 7-17}
\]

The original SELECT query now will involve above \( k \times p \) tables. The sequential timing model for such select query is shown in Eq. 7-18.

\[
T_{\text{seq} \_ p} = \frac{p^k \times \left(\frac{n}{p}\right)^k}{W} + \frac{p^k \times \left(\frac{n}{p}\right)^k}{W} = \frac{n^k + n \times p^{k-1}}{W} = O(n^k) \tag{Eq. 7-18}
\]

If we can evenly distribute those data partitions among a RAIDB system so that the distributed load on each server are close to each other, The corresponding parallel timing model can be written as Eq. 7-19.

\[
T_{\text{para} \_ p} = \frac{p^k \times \left(\frac{n}{p}\right)^k}{N \times W} + \frac{p^k \times \left(\frac{n}{p}\right)^k}{W} + \frac{p^k \times \left(\frac{n}{p}\right)^k}{W_{\text{net}}} = \frac{O(n^k)}{N \times W}, \quad W_{\text{net}} \text{ is the power of net} \tag{Eq. 7-19}
\]
The throughput speedup for such SELECT request on k partitioned tables can be shown as

Eq. 7-20

\[ Sp = \frac{T_{seq}}{T_{para \cdots p}} \approx N \]

Eq. 7-20 shows that near linear speedup can be achieved in the RAIDB system using data partition, when the target data and clients' request can be evenly distributed among all servers. The data size n should be much greater than the number of logical data partitions p.

### 7.3 Logical Partition and Virtual Partition in RAIDB System

Tuples in a horizontal partitioned object are divided into multiple separated **logical partitions**. These logical partitions can be distributed among multiple servers. Each logical partition has a unique data range. Any tuple in a partitioned object can only reside in one logical partition. Overlap between any two logical partitions is not permitted.

There maybe more than one logical partition resides on a database server. From database administrator’s viewpoint, all these logical partitions residing on the single server makes a **virtual partition**. With parallel synchronous replication technology in use, one tuple may reside in one virtual partition on Server A and also resides in another virtual partition on Server B. Under such scenario, we say the two virtual partitions overlap each other in that RAIDB system.
Figure 7-2 shows a typical SELECT query from TPC-E, which is designed to emulate typical transaction activities on OLTP system. This SELECT query is wrapped in a stored procedure for the convenience of programming. It searches the TRADE and TRADE_HISTORY tables to find up to 30 history rows that correspond with the 10 most recent trades executed by a specified customer account. Both TRADE and TRADE_HISTORY are Growing tables as we defined in section 3.3.3. Table TRADE is accessed twice in the query. The first scan on TRADE retrieves the latest 10 transactions under the specified account. The second scan on TRADE retrieves the latest 30 transaction records corresponding to the 10 most recent trades retrieved from the first scan.

Create Procedure [dbo].[spr_F2b_GetCustSum2]
(@acct_id dbo.IDENT_T)
AS
Begin

select top 30
    trade_id[] = T_ID,
    symbol[] = T_S_SYMB,
    qty[] = T_QTY,
    trade_status[] = ST_NAME,
    hist_dts[] = TH_DTS
from
    (select top 10 T_ID as ID
     from TRADE
     where
         T_CA_ID = acct_id
     order by
         T_DTS desc) as T,
    TRADE,
    TRADE_HISTORY,
    STATUS_TYPE
where
    T_ID = ID and
    TH_T_ID = T_ID and
    ST_ID = TH_ST_ID
order by
    TH_DTS desc
END
Go

--SET ICXP TPCE READSLIM ON 4300000006
Exec [dbo].[spr_F2b_GetCustSum2] 43000000006
--SET ICXP TPCE READSLIM OFF 4300000006

Figure 7-2 Example of Read-only Operation using READSLIM
For SELECT queries like the one in Figure 7-2, the RAIDB gateway can use the information wrapped in PRML statements to forward such queries to those proper servers where the data exist. Once the target server receives such queries, it may not have enough information from the query to decide which logical partition to scan. The whole virtual partition on that server will be scanned. Dividing data into finer logical partitions within one server will not increase the throughput speedup because all data in that virtual partition will be scanned no matter they are in one coarse-grain partition or multiple fine-grain partitions.

Since the virtual partition number equals the number of servers in a RAIDB system, we can use Eq. 7-6 and Eq. 7-7 to estimate the optimal logical partition number. With predetermined redundancy degree, the desired speedup value and load properties, we can get the number of server (N) needed in a RAIDB system. That is the optimal number of virtual partitions.

Keeping only one logical partition on each server may not be a good idea since skew may happen on both data distribution and load distribution. Keeping 10–20 logical partitions on each server can be helpful to address such skew issue. If the load on one server is much larger than the average load among all RAIDB servers, we can dynamically ship a few logical partitions to other servers to balance the load. Dividing data into finer partition within one server may not help too much about speedup for queries like the one in Figure 7-2. Therefore, the optimal logical partition number should be in the neighbor of 10 to 20 times of the number of virtual partitions.
7.4 Summary

This chapter presents the timing model for all four basic operations in a RAIDB system. The concept of virtual partition is discussed. It gives a method to calculate the optimal partition numbers and provides a solution to avoid data skew and load skew among data servers in a RAIDB system.
CHAPTER 8 EXPERIMENTAL RESULTS

In this chapter we present the test results of basic read and write operations on partitioned objects in a RAIDB system. The chosen tests includes: write-intensive operations on partition, general read operation on partition and super-linear speedup read operation on partition. These tests are based on the timing model discussed in previous chapter. A mixed transaction emulator is adapted from the standard database benchmark TPC-E to measure the overall throughput speedup of a RAIDB system.

8.1 Benchmark Database TPC-E

In this chapter the standard benchmark TPC-E will be used to measure the performance of a RAIDB system using Functional Horizontal Partitioning technology. TPC-E is a database schema designed to emulate typical OLTP workloads for online business. The database is required to be continuously available 24 hours a day, 7 days a week.

TPC-E benchmark includes 10 read-only and update intensive transactions (see Table 3-2) to simulate the activities in regular OLTP application environments. These transactions represent a breadth of system components associated with OLTP system, such as:

- Transaction integrity (ACID properties);
- Concurrent transactions compete on data access and update;
- Uniform and non-uniform data access through primary and secondary keys.
As we discussed in Chapter 3 (see Figure 3-11), the TPC-E database can be partitioned along the ID of customers. Eight tables are selected for horizontal partition because these tables all root from the customer table. Over 90% of write operations are related to these tables. All these tables belong to the “Growing table” we discussed in section 3.3.2. Table 3-3 shows the final partition schema and transaction footprint. For reader’s convenience, it is redrawn below.

![Table 3-3: Footprints of TPC-E Benchmark Transactions](image)

### Table 3-3: Footprints of TPC-E Benchmark Transactions

<table>
<thead>
<tr>
<th>Category</th>
<th>Table Name</th>
<th>Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Customer</td>
<td>ACCOUNT_PERMISSION</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>CUSTOMER</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>CUSTOMER_ACCOUNT</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>CUSTOMER_TAX_RATE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>HOLDING</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>HOLDING_HISTORY</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>HOLDING_SUMMARY</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>WATCH_ITEM</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>WATCH_LIST</td>
<td>RO</td>
</tr>
<tr>
<td>Broker</td>
<td>BROKER</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>CHARGE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>COMMISSION_RATE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>CASH TRANSACTION</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>SETTLEMENT</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>TRADE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>TRADE_HISTORY</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>TRADE_REQUEST</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>TRADE_TYPE</td>
<td>RO</td>
</tr>
<tr>
<td>Market</td>
<td>COMPANY</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>COMPANY_COMPETITOR</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>DAILY_MARKET</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>EXCHANGE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>FINANCIAL</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>INDUSTRY</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>LAST_TRADE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>NEWS_ITEM</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>NEWS_XREF</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>SECTOR</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>SECURITY</td>
<td>RO</td>
</tr>
<tr>
<td>Dimension</td>
<td>ADDRESS</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>STATUS_TYPE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>TAXRATE</td>
<td>RO</td>
</tr>
<tr>
<td></td>
<td>ZIP_CODE</td>
<td>RO</td>
</tr>
</tbody>
</table>

[Note]: RO = Read-Only, RW=Read-Write
8.2 Experimental Configuration

RAIDB Model 1001

<table>
<thead>
<tr>
<th>Database Server Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operating System</strong></td>
</tr>
<tr>
<td>Microsoft Server 2003 Service Pack 2</td>
</tr>
</tbody>
</table>

**Figure 8-1 General Experiment Configuration**
Figure 8-1 displays the basic experiment configuration for the new Functional Horizontal Partitioning technology on a RAIDB system. This RAIDB system includes three database servers and one separated workstation. All client transactions must go through the gateway. Multiple client threads are running simultaneously on the client terminal to simulate concurrent connections in real OLTP environment. Additional clients can be added if necessary.

The overhead of network delay can be ignored because all client transactions are balanced with mixture of intensive disk read/write and processor usage. The overhead on the middleware gateway is in the range of 10~50ms for each single query. For each client connection, it requires about 20 KB memories on the gateway, which can be easily fulfilled by a regular workstation or desktop PC. Previous experiments shows that the gateway running on a regular workstation can sustain up to 12,000 concurrent connections without noticeable performance deterioration on the gateway.

To avoid dynamic memory management overhead, all three database servers use fixed memory size for the database engine. In these experiments, the memory size for each database server engine is limited as 4GB to prevent all data been loaded into physical memory.

Multiple performance tuning methods, such as using index and separated disks for data/index/log for best optimization, have been applied to each database server.
8.3 Test Cases

This section discusses three test cases. Test case 1 represents typical read-only queries on a RAIDB system using Functional Horizontal Partitioning technology. Test case 2 represents typical write operation on a RAIDB system. Test case 3 use the Client Emulator engine from the TPC-E benchmark to test both the FHP technology and PRML parser developed in this dissertation.

8.3.1 Typical Read Operations

Figure 8-2 shows such a typical query submitted by a customer to retrieves the latest 50 trades related to the supported customer id (@c_id) and account number (@acct_id).

```sql
select top 50
    T_ID,
    T_DTS,
    ST_NAME,
    TT_NAME,
    T_S_SYMB,
    T_QTY,
    T_EXEC_NAME,
    S_NAME,
    EX_NAME
from
    dbo.TRADE,
    dbo.STATUS_TYPE,
    dbo.TRADE_TYPE,
    dbo.SECURITY,
    dbo.EXCHANGE
where
    T_C_ID  = @c_id    and
    T_CA_ID = @acct_id and
    ST_ID   = T_ST_ID  and
    TT_ID   = T_TT_ID  and
    S_SYMB  = T_S_SYMB and
    EX_ID   = S_EX_ID
order by  T_ID desc
```

Figure 8-2 SQL Code of Read-Only Test
If a Select query is frequently executed, corresponding index can be created to improve the query performance. If such queries can be evenly load-balanced among member servers, linear speedup is expected base on the performance modeling analysis in Chapter 7. Figure 8-3 shows the partition schema of TPC-E database partitioned by customer ID. The TPC-E database is partitioned into 12 logical partitions. These 12 logical partitions can reside on separated servers or reside on one single server. Each query in Figure 8-2 can be handled by a server where the target partition resides.

Figure 8-3 Experiment Configuration for Read-Only Test
Three cases are compared in this test. Case A serves as the baseline case where all queries are executed on one server. Case B uses two servers, and each server keeps six partitions. Case C uses three servers, and each server hosts four partitions. The gateway sitting between clients and database servers will distribute these queries based on the supported customer ID.

Figure 8-4 gives the throughputs of the three cases.

### Test Result of Typical Read Operations

<table>
<thead>
<tr>
<th>Thread Number</th>
<th>Single Server</th>
<th>Two Servers</th>
<th>Three Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Elapsed Time, s</td>
<td>Throughput</td>
<td>Elapsed Time, s</td>
</tr>
<tr>
<td>10</td>
<td>59.286</td>
<td>151.978</td>
<td>51.286</td>
</tr>
<tr>
<td>15</td>
<td>150.495</td>
<td>149.957</td>
<td>75.070</td>
</tr>
<tr>
<td>20</td>
<td>198.771</td>
<td>150.172</td>
<td>98.177</td>
</tr>
<tr>
<td>25</td>
<td>248.813</td>
<td>151.937</td>
<td>124.787</td>
</tr>
<tr>
<td>30</td>
<td>290.528</td>
<td>154.886</td>
<td>144.876</td>
</tr>
</tbody>
</table>

The results in Figure 8-4 verify the performance modeling analysis in Chapter 7: putting data partition on separated server can improve the read performance for RAIDB system.
Discussion:

The Functional Horizontal Partitioning technology makes a database system easy to be extended to hundred or even thousand of server nodes. Figure 8-5 displays a RAIDB system using Functional Horizontal Partitioning and a parallel synchronous replication database system without Functional Horizontal Partitioning. At the same data redundancy degree, the former can scale to more servers than the latter. The partitioned dataset in RAIDB can be smoothly shipped to other nodes without shutting down the service. Load can be easily balanced among member nodes. Server nodes with smaller dataset in the RAIDB system can be built from regular cost-efficient servers instead of those expensive and customized systems. Such RAIDB system is both flexible and extensible to the on-changing work load.
8.3.2 Write-Intensive Operations on Partition

Three groups of control experiments are designed to evaluate the performance of a RAIDB system when handling write-intensive operations. Figure 8-6 shows a stored procedure with write-intensive operations on the database partitioned by relational parallel partitioning.

```sql
Create Procedure [dbo].[spw_Intensive_Write_Test]
    (@Customer_id dbo.IDENT_T,  @now_dts  dbo.DATE_TIME)
AS
BEGIN
    Declare @trade_id     dbo.TRADE_T
    Declare  trade_list cursor for
        select top 100 T_ID
        from     dbo.TRADE              -- paritioned object
        where    T_C_ID = @Customer_id and T_DTS < @now_dts
        Order by T_ID ASC
    Open trade_list
    FETCH NEXT FROM trade_list into @trade_id
    WHILE @@FETCH_STATUS = 0
    BEGIN
        Update dbo.Trade
        Set    T_DTS  = @now_dts    -- paritioned object
        where  T_ID   = @trade_id
        Update dbo.Trade_History    -- paritioned object
        Set    TH_DTS  = @now_dts
        where  TH_T_ID = @trade_id
        Update dbo.Cash_Transaction -- paritioned object
        Set    CT_DTS  = @now_dts
        where  CT_T_ID = @trade_id
        Update dbo.Settlement       -- paritioned object
        Set    SE_CASH_DUE_DATE = @now_dts
        where  SE_T_ID   = @trade_id
        Update dbo.Holding          -- paritioned object
        Set    H_DTS  = @now_dts
        where  H_T_ID = @trade_id
        FETCH NEXT FROM trade_list into @trade_id
    END --end of list while
    CLOSE trade_list
    DEALLOCATE trade_list
End
```

Figure 8-6 Stored Procedure for Write-Intensive Test
The stored procedure in Figure 8-6 emulates intensive write actions initiated by a customer to update transaction records under his/her name. This stored procedure can be executed by following statement.

```
--SET ICXP TPCE WRITESLIM ON 4300000560
Exec dbo.spw_Intensive_Write_Test 4300000560, '2009-04-28'
--SET ICXP TPCE WRITESLIM OFF 4300000560
```

A unique customer ID is assigned to each thread. It is designed to avoid any deadlock among client sessions so that the whole database system can achieve its highest potential throughput. Similarly, a connection pool is used to reduce the overhead of building connections.

This experiment includes three test cases to measure the performance with different data redundancy and overhead of the new PRML parser. These three cases are:

Case I: Write-intensive without redundant data copy. The target dataset is partitioned into three partitions. Three servers are used in this case. Each server keeps an active partition. The overall system performance is compared to a system using only one server (see Figure 8-7).

Case II: Write-intensive with redundant data copy. This case is very similar to case I, except each server keeps two active partitions and the overall data redundancy for the whole RAIDB system is 2 (see Figure 8-9).

Case III: In this test we use only one database server. We first put the target dataset in its original non-partitioned table and let client program directly access this database server. Then we partition the dataset into multiple partitions and keep all partitions
in the same server. We put a gateway in front of the database server and measure the performance again (see Figure 8-11). This test is designed to estimate the overhead of such RAIDB system.

Case I: Write-intensive Test without Redundant Data Copy (R=1)

![Figure 8-7 Experiment Configuration for Write-Intensive Test (1)](image)

**Figure 8-7 Experiment Configuration for Write-Intensive Test (1)**

Figure 8-8 displays the throughput results of Case I. It shows that the average throughput speedup for 30~150 concurrent write-intensive connections is 2.9, which is very close to the theoretical linear-speedup value (3).
Test Result of Case I

<table>
<thead>
<tr>
<th>Thread Number</th>
<th>Single Server</th>
<th></th>
<th>Three Servers</th>
<th></th>
<th>Throughput</th>
<th></th>
<th>Throughput</th>
<th></th>
<th>Throughput</th>
<th></th>
<th>Throughput Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Elapse Time(s)</td>
<td>Throughput</td>
<td>Elapse Time(s)</td>
<td>Throughput</td>
<td>Speedup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>1486.5</td>
<td>204.6</td>
<td>530.5</td>
<td>565.5</td>
<td>2.784</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>2674.3</td>
<td>224.4</td>
<td>965.6</td>
<td>621.4</td>
<td>2.770</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>8523.8</td>
<td>184.0</td>
<td>2153.8</td>
<td>557.2</td>
<td>3.029</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>8037.6</td>
<td>186.6</td>
<td>2806.4</td>
<td>534.5</td>
<td>2.864</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Average throughput speedup 2.900

Figure 8-8 Experiment Result of Write-Intensive Test (1)

Case II: Write-intensive Test with Redundant Data Copy (R=2)

Figure 8-9 Experiment Configuration for Write-Intensive Test (2)
In test A of Case II, there are three servers and two copies of data. Each server keeps two active partitions. Therefore, the workload on each server will be two times of the workload we see in previous case. There are two servers in test B; each of them keeps a full data copy.

### Test Results of Case II

<table>
<thead>
<tr>
<th>Thread Number</th>
<th>Case A: 3 Servers</th>
<th>Case B: 2 Servers</th>
<th>Throughput Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapse Time,s</td>
<td>Throughput</td>
<td>Elapse Time,s</td>
<td>Throughput</td>
</tr>
<tr>
<td>30</td>
<td>1095.2</td>
<td>273.9</td>
<td>1450.0</td>
</tr>
<tr>
<td>60</td>
<td>2086.4</td>
<td>297.8</td>
<td>3023.7</td>
</tr>
<tr>
<td>90</td>
<td>3075.2</td>
<td>292.7</td>
<td>4710.5</td>
</tr>
<tr>
<td>120</td>
<td>4280.7</td>
<td>280.3</td>
<td>6385.9</td>
</tr>
<tr>
<td>150</td>
<td>5291.5</td>
<td>288.5</td>
<td>7961.2</td>
</tr>
</tbody>
</table>

Average throughput speedup: **1.456**

*Figure 8-10 Experiment Result of Write-Intensive Test (2)*

Figure 8-10 displays the throughput results of Case II. The overall throughput of the three servers drops almost half from Case I. This is because the workload distributed to each server is doubled. The throughput of case B (full data copy) doesn’t change much after adding one new server. It is because each thread is working on a different customer account, there is not wait or deadlock among working threads. The average throughput speedup is 1.456, which is nearly half of the throughput speedup of Case I.
Case III: RAIDB System Overhead

Figure 8-11 displays the experiment configuration of case III. Test A is the baseline test. The database server can be directly accessed by client testing program. The data in test A is not partitioned. It represents the regular practice of OLTP business. The test B is the simplest RAIDB system. There is only one server and only one gateway in this RAIDB system. The data is partitioned into multiple logical partitions.

Test Results of Case III

<table>
<thead>
<tr>
<th>Thread Number</th>
<th>Elapse Times, s A</th>
<th>Elapse Times, s B</th>
<th>Overhead, % 100%(B-A)/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>1421.3</td>
<td>1466.3</td>
<td>3.166</td>
</tr>
<tr>
<td>60</td>
<td>2710</td>
<td>2574.3</td>
<td>-1.317</td>
</tr>
<tr>
<td>90</td>
<td>4529.7</td>
<td>4595.5</td>
<td>1.453</td>
</tr>
<tr>
<td>120</td>
<td>6455.8</td>
<td>6523.3</td>
<td>1.046</td>
</tr>
<tr>
<td>150</td>
<td>8217.7</td>
<td>8037.6</td>
<td>-2.192</td>
</tr>
</tbody>
</table>

Average 0.431

Table 8-1 Experiment Result of Overhead Test
Table 8-1 shows the test results of Case III. The average overhead is less than 0.5%. Besides, the overhead values at Thread=60 and Thread=150 are less than 0. It indicates that the error incurred by network fluctuation and system is greater than 2%. Experiments by Shi also show that such gateway overhead is less than 10~50ms for each client query [1]. Considering the existence of such error, this result indicates that the overhead of gateway and the new Functional Horizontal Partitioning technology is ignorable for write-intensive operations.

8.3.3 Mixed Cases

In previous two sections we measured the performance of a RAIDB system integrated with Functional Horizontal Partitioning technology. Test results shows that such RAIDB system is both flexible and scaleable. In this section we will use the standard benchmark TPC-E to perform a full functionality test and measure the performance of a RAIDB system integrated with Functional Horizontal Partitioning technology.

Four test cases are designed to measure the performance of a RAIDB system integrated with FHP technology, see Figure 8-12. Case A serves as a baseline case, which gives the performance of a single database system. Case B and C are systems using parallel synchronous transaction replication technology not integrated with Functional Horizontal Partitioning technology. In these two cases, each server maintains a full data copy. Case D shows the experiment configuration of the RAIDB integrated with FHP technology. This RAIDB system emulates a real business environment. It includes three database servers and
a DBx gateway. The target database is partitioned into 12 logical partitions. Among the three database servers, server A hosts all 12 partitions and behaves as the Full Server, server B and C work as Slim Servers and each server hosts 6 logical partitions. The overall data redundancy is 2. Client queries will be balanced among all member nodes. The Full Server handles complicated client queries which cannot be handled by slim server.

![Diagram](image1)

**Figure 8-12 Experiment Result of TPC-E Benchmark Test**

![Diagram](image2)

**Table 8-2 Experiment Result of TPC-E Benchmark Test**

<table>
<thead>
<tr>
<th>Threads*</th>
<th>Baseline N=1, R=1</th>
<th>Case B N=2, R=2</th>
<th>Case C N=3, R=3</th>
<th>Case D N=3, R=2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Elapsed Time, s</td>
<td>Elapsed Time, s</td>
<td>Elapsed Time, s</td>
<td>Elapsed Time, s</td>
</tr>
<tr>
<td>10</td>
<td>1956</td>
<td>1240</td>
<td>959.0</td>
<td>750.5</td>
</tr>
<tr>
<td>15</td>
<td>2952</td>
<td>1798</td>
<td>1448</td>
<td>1106</td>
</tr>
<tr>
<td>20</td>
<td>3952</td>
<td>2408</td>
<td>1873</td>
<td>1488</td>
</tr>
<tr>
<td>25</td>
<td>5041</td>
<td>3036</td>
<td>2418</td>
<td>1987</td>
</tr>
<tr>
<td>30</td>
<td>6044</td>
<td>3569</td>
<td>3005</td>
<td>2195</td>
</tr>
</tbody>
</table>

* Each thread executes 1000 mixed transactions
Table 8-2 shows the performance result of the four test cases. The result of baseline case A are compared with other cases to calculate the overall throughput speedup of each system. Comparing the results of Case B and C, we can see that the marginal contributions of throughput speedup decreases. It verifies the analysis in Chapter 7. Case D shows the potential capability of FHP technology, the speedup is better than its counterpart in Case B and C. It reveals the possibilities of delivering both high performance and high availability by extending a database system to multiple servers behind parallel replication and Functional Horizontal Partitioning technology.

8.4 Summary

Three groups of experiments are designed to evaluate the performance of a new RAIDB system integrated with the new Functional Horizontal Partitioning technology. These experiments demonstrate the scalability of a RAIDB system integrated FHP technology for both read-only query and write-intensive operation.
CHAPTER 9 CONCLUSION

This thesis presents a new Functional Horizontal Partitioning technology to allow a resource-efficient implementation and its supporting application programming interface. It leverages two breakthroughs on synchronous transaction replication and Stateless Parallel Processing by Justin Y. Shi and his student Suntain Song. Partition candidates are chosen basing on transaction footprints and table characteristics. This partitioning is designed for transparent non-intrusive deployments. Such schema keeps the autonomy of each member server and minimizes the message exchange among servers. A very large OLTP database system integrated with parallel synchronous replication system and this new Functional Horizontal Partitioning technology can evenly distribute most transaction to member servers. With the support of parallel synchronous replication technology and FHP technology, users can smoothly extend the cluster size to hundred or even thousand nodes without transaction loss and the overall system recover time is kept less than 60 seconds, independent of data size.

Using this proposed approach, users can use simple PRML tags to send partition-oriented commands to the middleware gateway, which can automatically forward user’s SQL statements to the proper target server(s). These partition-oriented commands wrapped in PRML tags are automatically extracted by the PRML parser and used to generate parallel processing. All PRML tags are transparent to regular database engine. It makes the code
generated for the RAIDB system be compatible to a system not integrated with parallel replication gateway.

The performance modeling shows that linear speedup can be achieved for both read-only query and write-intensive operation. It also gives the method to estimate the optimal cluster size and the number of data partitions.

Three groups of experiments are designed to measure the throughput speedup of a RAIDB system integrated with FHP technology. Test results show that this new RAIDB architecture can deliver linear speedup for most client operations. The industrial standard benchmark is also used to test the functionality and performance of the new RAIDB architecture. It shows that the new RAIDB architecture can deliver better scaleup better and higher speedup than regular parallel synchronous replication system without using such Functional Horizontal Partitioning technology.

9.1 Summary of Contributions

This thesis is the first effort to solve database replication on a self-resilient database cluster with customizable data redundancy schema. The envisioned contribution of this research include

- Design of a customizable Functional Horizontal Partitioning strategy which allows automatic dataset partition, data distribution and seamless (non-stop) recovery.
• Design of a partition replication markup language (PRML)
• Provide Performance Modeling for optimizing N, P and R.
• Provide Performance Modeling and Analysis for PSR architecture integrated with FHP schema.
• Implementation of a new parser for the new partition replication markup language.
• Design and implementation of an application programming interface (API) translating user queries using virtual global schema into parallel queries targeting multiple physically partitioned schemas.

9.2 Future Work

We have shown that the new Functional Horizontal Partitioning technology is simple yet powerful for very large OLTP database applications. It prevents the overall throughput from deteriorating due to increased workload and data synchronization overhead.

With this new architecture, high performance and high availability are both achievable. Further researches in this direction include:

• Enhance the PRML parser to automatically analyze complex client queries so that these queries can be divided and executed in parallel. This can remove the bottleneck on the full server and increase the parallelism of query processing.
• Develop Multi-Dimension Functional Horizontal Partitioning Methodology. Currently, the Functional Horizontal Partitioning technology only supports one dimension partitioning. Developing Multi-Dimension FHP technology can provide more options for database developer. It can also provide more load balance options for the PRML parser.

• Enrich PRML tags to further increase their expressive power. Interactive tools may be built to assist program tagging.

• Automate Partition support. It has been shown that the analysis on transaction footprints and database schema can effectively guide the design of Functional Horizontal Partitioning strategy. It is possible to develop tools that an automatically analyze and partition heavily accessed and fast growing tables.
REFERENCE

[1]. Justin Y. Shi, “DBx SQL Server Cluster Middleware”,

Management of Data archive, Proc. 1996 ACM SIGMOD international conference on
Management of data table of contents, Page 173 - 182 , Montreal, Quebec, Canada, 1996

Symposium, Page 287-290

[4]. D. DeWitt and J. Gray. “Parallel database systems: The future of high

[5]. IBM, “Enterprise replication: a high-performance solution for distributing and
sharing information”,
August 2003


[7]. Plattner, Christian "Ganymed: scalable replication for transactional web
applications", Middleware Conference; Vol. 78, Proceedings of the 5th
ACM/IFIP/USENIX international conference on Middleware table of contents, Page.
155 - 174, Toronto, Canada, 2004

[8]. Yair Amir, "From Total Order to Database Replication", Proceedings of the 22 nd
International Conference on Distributed Computing Systems (ICDCS'02), Page 494 - 503,
2002

[9]. Xiaohu Yang, "Data Based Application Partitioning and Workload Balance in
Distributed Environment", Proceedings of the International Conference on Software
Engineering Advances, Page 43 - 48, 2006

[10]. Peter Scheuermann, "Data Partitioning and Load Balancing in Parallel Disk


[21]. Justin Yuan Shi and Eagles Song, DBx SQL Server Cluster Middleware online book

[22]. The Professional Learning Centre of University of Toronto, Taxonomy Guide http://plc.fis.utoronto.ca/tgdemo/Glossary.asp


[24]. Oracle 11g online library, http://download.oracle.com/docs/cd/B28359_01/server.111/b28313/usingpe.htm#i1007101

BIBLIOGRAPHY


Amir, Yair "From Total Order to Database Replication", Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), Page 494 - 503, 2002


Hua, Kien A. “Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning”, Proceedings of the 17th International Conference on Very Large Data Bases, Pages: 525 - 535, 1991

Lima, Alexandre A. B. "Adaptive Virtual Partitioning for OLAP Query Processing in a Database Cluster", Brazilian Symposium on Databases, 2004


Oracle 11g online library, http://download.oracle.com/docs/cd/B28359_01/server.111/b28313/usingpe.htm#i1007101


Professional Learning Centre of University of Toronto, Taxonomy Guide http://plc.fis.utoronto.ca/tgdemo/Glossary.asp


Shi, Justin Yuan DBx SQL Server Cluster Middleware online book

Sun, Feijian “Automatic Program Parallelization using Stateless Parallel Processing Architecture”, Ph. D. dissertation of Temple University, 2004
