P*TIME: Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload

Sang K. Cha and Changbin Song

Transact In Memory, Inc.
1600 Adams Drive
Menlo Park, CA 94025
USA

Seoul National University
School of Electrical Eng. and Computer Science
Kwanak P.O.Box 34, Seoul 151-600
Korea

{chask, tsangbi}@transactinmemory.com

Abstract

Over the past thirty years since the system R and Ingres projects started to lay the foundation for today's RDBMS implementations, the underlying hardware and software platforms have changed dramatically. However, the fundamental RDBMS architecture, especially, the storage engine architecture, largely remains unchanged. While this conventional architecture may suffices for satisfying most of today's applications, its deliverable performance range is far from meeting the so-called growing "real-time enterprise" demand of acquiring and querying high-volume update data streams cost-effectively.

P*TIME is a new, memory-centric light-weight OLTP RDBMS designed and built from scratch to deliver orders of magnitude higher scalability on commodity SMP hardware than existing RDBMS implementations, not only in search but also in update performance. Its storage engine layer incorporates our previous innovations for exploiting engine-level micro-parallelism such as differential logging and optimistic latch-free index traversal concurrency control protocol. This paper presents the architecture and performance of P*TIME and reports our experience of deploying P*TIME as the stock market database server at one of the largest on-line brokerage firms.

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

1. Introduction

1.1. Demand for new OLTP DBMS architecture

Thirty years have passed since the system R and Ingres projects started to lay the foundation for today's RDBMS implementations [1]. Over this period, Moore's law has driven CPU processing power and memory capacity to grow million times, or 60% per year, respectively. The underlying software platform also changed significantly. Most operating systems now support virtually infinite address for 64-bit CPUs. The POSIX lightweight thread package enables efficient utilization of high-performance commodity multiprocessor hardware.

However, despite these dramatic underlying changes, the fundamental architecture of a single RDBMS instance largely remains unchanged. Even though data and indexes are cached in large buffer memory, they are managed as disk-resident structures. The heavyweight process architecture, which incurs high context switching overhead among multiple processes involved in executing a transaction, is still dominant [2]. This disk-centric heavyweight RDBMS architecture with the multi-million-line code base evolving over decades is inevitably subject to growing impedance mismatch with the underlying hardware capability. Recent research on L2-cache-conscious database structures and algorithms such as [3][4][5][6][7] addresses a crucial aspect of this mismatch that was not taken into consideration when existing RDBMS implementations, whether disk-centric or in-memory, were architected and implemented.

While the conventional disk-centric RDBMS architecture may suffice to serve search-dominant applications, the number of applications demanding the performance beyond the practical limit of today's RDBMS implementations is growing. Such applications typically deal with update-intensive stream workload, and are often called "real-time enterprise" applications by the business community. Some representative examples are:

- Stock market data stream in financial services.
- Call detail record (CDR) and network monitoring data streams in communication carriers: Especially challenging to cope with is the increasing CDR volume with the support of packet-granularity billing.
- Click streams in large portals.
- Update streams in on-line travel services.
- RFID data streams in supply chain management and retail.
- Traffic data management.

With the update transaction processing capability of typical RDBMS implementations limited to a few hundred TPS (transactions per second) on commodity SMP hardware, many painful hacks are commonly used in practice for handling update-intensive stream workload.

- Use of low-level ISAM files instead of RDBMS at the cost of giving up the high-level declarative SQL and ACID transaction quality.
- Heavy dependence on message queue systems placed in front of OLTP database at the cost of increased latency and capital expenditure.
- Excessive database partitioning and tuning on top of heavy hardware investment.
- Application-level batch processing with the risk of data loss and at the expense of application complexity.

1.2. P*TIME with storage engine innovations

Designed and built from scratch starting in 2000 with about 50 man-year effort, P*TIME is a new, memory-centric lightweight OLTP RDBMS that delivers up to two orders of magnitude higher scalability on commodity SMP hardware than existing RDBMS implementations.

P*TIME manages performance-critical data and indexes in the memory of a single multithreaded process. This architectural framework resulted from our prior experience of developing and benchmarking an in-memory storage engine over several years, which became the basis of multiple in-memory DBMS products in commercial production at Korean telecom and financial institutions [8][9][10]. However, the internal storage engine implementation details and capability fundamentally differ from those of its predecessor and other first-generation in-memory DBMS implementations such as [11] in following ways:

- Highly scalable durable-commit update transaction processing performance up to 140K TPS on a single non-partitioned physical table residing in a commodity 4-way 64-bit PC server.
- Highly scalable fast database recovery: Recovering a database of several gigabytes in memory takes only a little over 1 minute.
- Superior multiprocessor scalability: By eliminating the well-known index locking bottleneck that limits the multiprocessor scalability of the first-generation in-memory or the memory-cached disk-centric database, P*TIME can execute 1.4M concurrent search TPS on a 4-way 64-bit PC server.
- Ability of dealing with time-growing database through transparent management of the aging portion in disks.

From the interview with major telecom and financial institutions that have deployed or attempted to deploy the first-generation in-memory DBMS technology for mission-critical applications, we have learned that the lack of these capabilities has led to the disappointment with the technology and eventually the substantial scale-down of planned deployments or the project cancellation in some cases. As a specific example of the technology disappointment, restarting a 50GB in-memory billing database system at a major Korean wireless carrier takes four hours on HP Super dome machine. This long recovery time is unacceptable even with hot-standby database replication.

Enabling the above differentiated set of P*TIME capabilities are our own storage-engine-level innovations that exploit micro parallelism on today’s shared-memory multiprocessor (SMP) hardware with multi-GHz CPUs, large memory, and a number of inexpensive disks. Differential logging, which enables fine-grained parallelism in logging and recovery of memory-centric databases [12], and optimistic latch-free index traversal (OLFIT) concurrency control, which maximizes parallel concurrent access to index nodes on SMP machines [5], are two representative innovations embedded in P*TIME to exploit such micro parallelism.

Differential logging uses bitwise XOR for undo and redo of database changes, each of which is captured as bitwise XOR difference between the after and before images of a fine-grained memory location. It minimizes the log volume to flush to the secondary storage while enabling fully parallel processing of an arbitrary number of differential log record streams independent of serialization order both during run time and recovery time. With each log record stream mapped to a physical disk, this means that the more log disks are added to a P*TIME database system, the shorter becomes the time to recover a database in memory and the higher durable-commit update transaction processing performance can be delivered.

The OLFIT defines an L2-cache-conscious concurrent tree index access protocol focused on minimizing node latch and unlatch operations. In an SMP environment, these operations incur excessive coherence L2 cache misses in reading or writing nodes, especially, upper ones, because the control information of an index node updated by one processor is highly likely to be updated by another

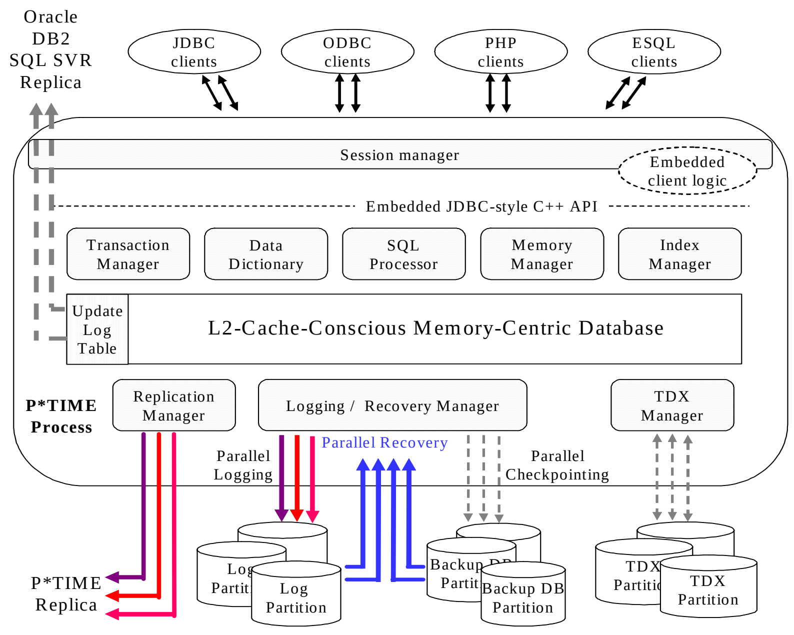


Figure 1. P*TIME Architecture

processor next time [5]. The coherence cache misses caused by index latching and unlatching is known for the major source of limiting the multiprocessor scalability of conventional in-memory or memory-cached disk-centric database systems. P*TIME, embedding the OLFIT protocol, enables multiple processors to access most index nodes concurrently without latching. For this reason, P*TIME shows almost linear multiprocessor scalability for indexed search, comparable to the scalability with no concurrency control. With minimized index node latching, P*TIME shows highly scalable multiprocessor index performance even for 100% update workload.

P*TIME currently supports SQL 92 with some extensions and standard RDBMS APIs such as ODBC/JDBC and a JDBC-style C++ API for building multi-tier and embedded applications, respectively. With its lightweight session management, P*TIME can sustain a thousand concurrent ODBC/JDBC connections without performance degradation.

P*TIME has been successfully in production since November 2002 as the stock market database server at Samsung Securities in Korea, one of the world-largest on-line stock brokerage companies that serves 50K – 60K concurrent on-line traders. P*TIME-based stock market database server processes 4 million trading messages per day with 2.6 million trading messages over the six hour window for the real-time update of the market database through SQL/ODBC API. This real-time database update consumes about 10% of the CPU power of 6-way 450Hz HP server and the rest is available for concurrent processing of up to 20K SQL query transactions per second per machine for the users who query the database through one of the best developed broadband and wireless infrastructures in the world. Our laboratory experiment shows that P*TIME is capable of processing up to eight times as fast as the peak-time arrival rate of real trading messages while reserving 70% of CPU power for concurrent query processing on a 4-way 700MHz PC server.

As another evidence for its industrial strength, porting a major enterprise software vendor's application with about 500 tables took only about a week. P*TIME is also being deployed in communications carriers, government agencies, and RFID-based supply chain management systems as reliable high-performance lightweight OLTP database systems.

1.3. Contribution of this paper

This paper presents the architecture and performance of P*TIME and reports our experience of deploying P*TIME as the stock market database server.

The major contribution of this paper is to show existentially that a new, carefully engineered memory-centric OLTP DBMS with the focus of exploiting engine-level micro parallelism can support the challenging performance requirement of update-intensive stream OLTP workload cost-effectively. Our previous work of differential logging and OLFIT concurrency control protocol plays critical roles in achieving up to two orders of magnitude difference in performance scalability combined with many implementation optimizations to minimize unnecessary overhead.

Our contribution is complimentary to the recent progress in the stream data management research focusing on the same target application domains. While the stream data management research focuses on the incremental, adaptive on-line analysis of stream data through a data flow network of operators, queues, and synopsis [13], our focus is on the high-performance storage and ad hoc query of update data streams addressing the needs of mission-critical enterprise applications that cannot tolerate any loss of data or continuity of service. The new TelegraphCQ implementation approach of starting from PostgresSQL instead of expanding its early Java implementation supports the need for a powerful storage engine for building practical stream data management systems [14].

This paper is organized as follows. Section 2 presents P*TIME architecture and its components. Section 3 presents the performance scalability of P*TIME storage engine with a brief description of the experimental measurement environment. Section 4 describes the challenges and experience of deploying P*TIME as the stock market database server. Section 5 concludes this paper.

2. P*TIME Architecture

P*TIME is a fully functional RDBMS. Figure 1 shows P*TIME architecture. Rounded boxes represent essential DBMS functional modules, and ovals represent applications.

2.1. Overall design goal

Based on our several years of experience of implementing and benchmarking a multithreaded in-memory DBMS since early 1990's [8][9][10], we designed P*TIME architecture with two major goals: maintaining a compact code base structure for the ease of code changes, and maximization of micro parallelism for exploiting the ever-increasing hardware capability.

To meet the first goal, C++ is chosen as the implementation language to structure the P*TIME code base as a collection of C++ classes, and the C++ template feature is used extensively. To meet the second goal, innovations are made to most of DBMS functionality layers, starting from the most fundamental layer of logging and recovery. In addition, the detailed implementation is guided by a derived guideline of avoiding unnecessary L2 cache misses and minimizing context switches, which also incur substantial L2 cache misses.

Our current focus of P*TIME on micro parallelism does not mean that we are excluding the natural extension of P*TIME exploiting macro parallelism on a distributed grid of P*TIME instances. In fact, to exploit the cost advantage of commodity SMP boxes, P*TIME provides the industry standard XA interface for supporting distributed transactions on partitioned databases, and the fast active-active database replication based on asynchronous fine-grained log propagation.

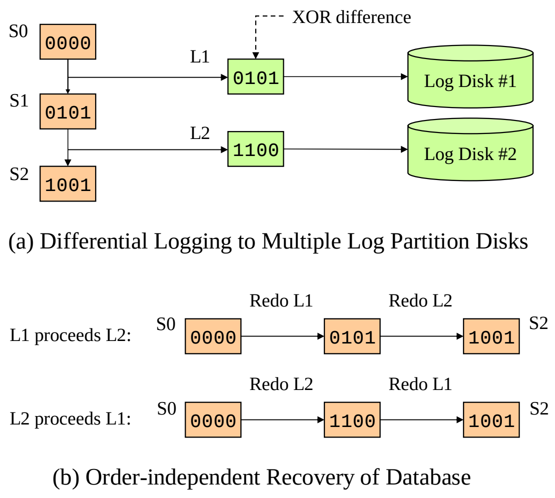


Figure 2. Inherent Parallelism of Differential Logging

2.2. L2-cache-conscious in-memory database

P*TIME manages performance-critical data and indexes primarily in the memory of a single multithreaded process. For the data, memory is divided into a set of pages, each of which contains multiple homogeneous slots for holding records, large variable-length fields, or pieces of BLOB data. A container is defined to manage a list of homogeneous pages. By default, a table is mapped to a container. Optionally, a table may be mapped to multiple containers, each of which stores a vertical partition of a table for efficient column-wise scan.

In P*TIME, indexes are by default managed as non-persistent structures supporting isolation and rollback and are rebuilt in parallel during the database restart process. P*TIME implements CPU-optimized hash and B+-tree index structures with direct memory addressing. In implementing B+-tree, we optimized the node layout and search and insertion procedures to minimize L2 cache misses. We chose B+-tree instead of CSB+-tree, the well-known cache-conscious B+-tree ([3]), because the optimized version of B+-tree performs better than CSB+-tree in the overall performance for the update-intensive workload.

2.3. Fine-grained parallel differential logging and recovery

P*TIME supports ACID transactions by storing every update log, first in an in-memory log buffer, and eventually in one of log partition disks. To recycle the log disk space and to shorten the database recovery time, P*TIME uses the parallelized version of fuzzy checkpointing. Dirty in-memory database pages are occasionally flushed in parallel to backup partition disks without interrupting transaction processing.

Compared with existing in-memory or disk-centric RDBMS implementations, P*TIME takes a very different architectural approach to logging. Based on the fine-grained differential logging of updates, P*TIME first minimizes log volume for maximal utilization of CPU processing power and memory and IO bandwidth. Dynamic selection of field-level or record-level logging represents the degree of P*TIME optimization for log volume minimization. Even for the same logging granularity, differential logging stores the XOR difference between "before" and "after" images and thus reduces the log volume by almost half compared with the conventional "before/after image" logging such as ARIES [15] [16]. Compared with the block-level "before/after image" logging implementations, the fine-grained differential logging of P*TIME reduces the log volume by an order of magnitude without sacrificing the recovery performance.

Figure 2 illustrates the inherent parallelism of differential logging as two serial transactions change a data item from S0 to S2. When the first transaction T1 changes the data from "0000" to "0101", the log record L1 with the XOR difference "0101" is flushed to the log disk #1 and T1 commits. When the second transaction T2 changes the data from "0101" to "1001", the log record L2 with the XOR difference "1100" is flushed to the log disk #2, and T2 commits. At this point, if the system crashes, the database is restarted by initializing the data item with the backup image S0 ("0000"). Differential logging uses the bit-wise XOR operation as redo and undo. Figure 2 (b) shows that the log records in two log disks can be processed in parallel because the associativity and commutativity of XOR enables the correct recovery of the state S2 can be recovered independent of the order of applying L1 and L2. Note that if the system crashed before the transaction T2 writing the commit record to the disk #2, T2 can be undone by applying L2 to S2 after doing all redo operations. Alternatively, the single pass recovery is possible by scanning the log files backward and applying only log records of committed transactions.

Parallel nature of differential logging further multiplies the gain in logging performance with the number of log partition disks, making it possible for P*TIME to deliver up to two orders of magnitude higher scalability in durable-commit update performance than existing RDBMS implementations. P*TIME supports hot spot updates by allowing a transaction to proceed to access the updates of a proceeding transaction waiting for the commit record to be flushed to log disks.

The scalability of durable-commit update performance is essential for the mission-critical real-time enterprise applications that cannot tolerate any loss of data. For less mission-critical applications, P*TIME also supports the so-called "deferred-commit" update mode, with which the system does not wait for the commit record to be flushed to the log disk to issue the commit signal to applications. In section 3, we shall demonstrate that the engine-internal performance of P*TIME in both durable-commit and deferred-commit modes is superior to the reported deferred-commit update performance of a popular in-memory DBMS implementation on a comparable hardware platform [17].

P*TIME supports high availability first with fine-grained parallel recovery of in-memory database. Breaking the common misconception that parallel logging lengthens database recovery time because of the overhead of sorting multiple log streams by serialization order, which is true for most conventional RDBMS implementations, the inherent parallelism of differential logging enables fast recovery of P*TIME database, scalable with the number of log and backup partition disks. Differential logging even permits simultaneous processing of multiple log partitions and backup database partitions. In addition to fast database recovery, P*TIME supports active-active log-based N-way asynchronous and synchronous replication to meet the high availability requirement of mission-critical applications.

2.4. Concurrency control

Another distinguishing feature of P*TIME is the OLFIT index concurrency control [5], which minimizes expensive coherence L2 cache misses incurred by conventional, latch-based index locking protocols in the SMP environment. The OLFIT scheme, based on the optimistic assumption that the conflict is rare and even if it occurs, can be resolved by retrying the node access, is designed to avoid latching and unlatching operations as much as possible. With OLFIT maximizing hardware-level parallelism in concurrent index node access, P*TIME does not suffer from the well-known index locking bottleneck of existing in-memory or memory-cached disk-centric RDBMS implementations. Since the effectiveness of the OLFIT depends on how well it is implemented, P*TIME uses assembly language to implement its key primitives to make sure that unnecessary L2 cache misses do not occur.

For the concurrency control of base tables, P*TIME implements multi-level locking and supports all four isolation levels of SQL.

2.5. Transparent Disk eXtension

For the time-growing data such as stream data, it is not feasible to keep the entire table in memory. P*TIME TDX (Transparent Disk eXtension) transparently migrates aging or infrequently accessed portion of a table to TDX partitions on disk. Each TDX partition is a self-describing indexed data set which supports compression and direct SQL access. TDX manager pins a TDX partition in memory when it is frequently accessed.

2.6. Application binding

On top of P*TIME core modules lies the SQL processor which includes a cost-based query optimizer and a query plan execution engine. This SQL processor is accessed through the standard programming interfaces such as ODBC, JDBC, ESQL, PHP, and a JDBC-style embedded

C++ API.

To protect the database system from application errors, application processes are completely decoupled from the multithreaded P*TIME database server. This differentiates P*TIME from the more tightly coupled heavyweight process architecture such as TimesTen, where multiple application processes directly access the database and the lock information in the shared memory. While this tightly coupled heavyweight process architecture has the advantage of eliminating client/server communication overhead for the clients running on the same machine with the database, it runs the risk of corrupting the database or blocking legitimate access by other processes because of locks unreleased by hanging applications. Furthermore, the gain in client/server communication disappears for the remote applications.

P*TIME also supports tight coupling of P*TIME with application logic but takes a different approach of providing an embedded C++ API. An application logic of accessing P*TIME database through a collection of JDBC-style C++ classes can be embedded inside a P*TIME server. This is useful for building P*TIME-embedded tools such as LDAP server.

To maximize multi-tier application performance, P*TIME transparently supports transaction group shipping (TGS) between P*TIME server and multi-threaded application servers. This feature reduces the number of interactions by shipping temporally adjacent independent transaction requests or responses in a group between P*TIME server and application servers.

2.7. Heterogeneous database integration interface

To facilitate the integration with heterogeneous RDBMS implementations, P*TIME provides an elegant interface called update log table whose entries representing recent update, insert, or delete on individual records can be accessed and deleted in SQL.

Compared with the common approach of vendor-specific, black-box synchronization functionality, this open-ended API enables application developers to implement arbitrary application-specific update propagation semantics to heterogeneous databases. With this feature, P*TIME can function as the transaction processing front-end to existing RDBMS implementations.

3. Performance Scalability

To measure the internal OLTP performance scalability of P*TIME, we embedded a simple benchmark logic inside the P*TIME server using its JDBC-style embedded C++ API. Workload clients are emulated by embedded connection objects iterated by several worker threads, which carry out actual transaction execution.

The test database consists of a single non-partitioned table of 8 million records that we adopted from a telco database. Since we are interested in measuring the scalability limit, we have intentionally avoided the creation of multiple tables or multiple partitions, which may distribute the internal contention pressure. The standard TPC benchmark was not adequate first because it is not intended for the OLTP of high-volume stream data, and secondly because the specification includes preset parameters such as think time that limit the maximum deliverable performance given a database size.

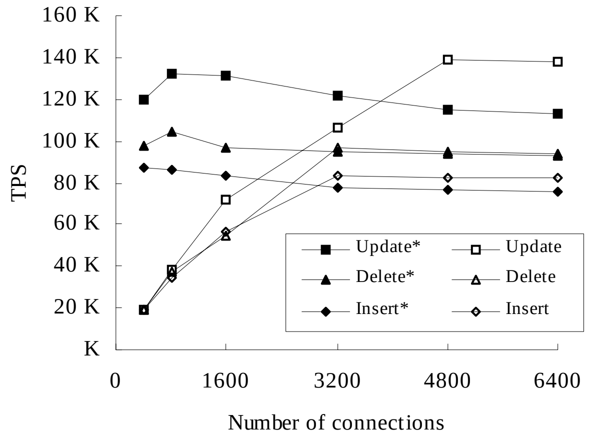


Figure 3. Update/Insert/Delete Performance with Varying Number of Connections
(The symbol * denotes that on-disk write cache is enabled.)

Each record of our test database is 168 bytes long. A hash index is built on its primary key field of BIGINT type. While we created the hash index in this experiment, the result is more or less the same for the B+-tree supporting range queries. The size of the initial database is about 1.4GB, including 100MB for the non-persistent hash index.

Each tested transaction type contains a single operation which is one of the following:

- Search a record with a given primary key and return a BIGINT-type column of the selected record.
- Update a BIGINT-type column of the record matching a given primary key.
- Insert a record.
- Delete a record matching a given primary key.

The experiment was mainly conducted on a Compaq ML 570 server running UNIX, with four 700MHz Xeon CPUs, each with 2MB L2 cache, 100MHz front-side bus, 6GB PC-100 SDRAM, Ultra 160 SCSI card, and several 7200 rpm SCSI disks. Each disk has 300KB of on-disk write cache (WC) for track buffering.

3.1. Update/Insert/Delete performance

From the durability perspective, we have two options in measuring the durable-commit update performance:

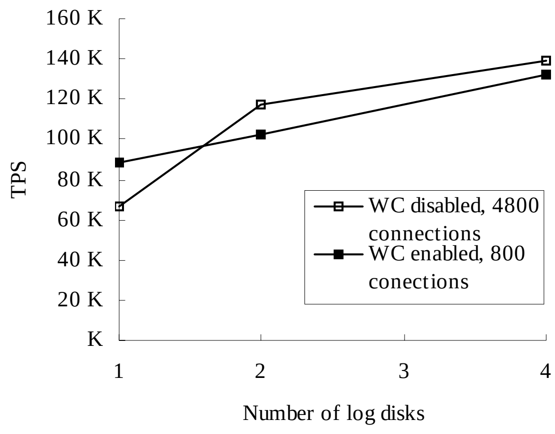


Figure 4. Update Scalability with Varying Number of Log Disks

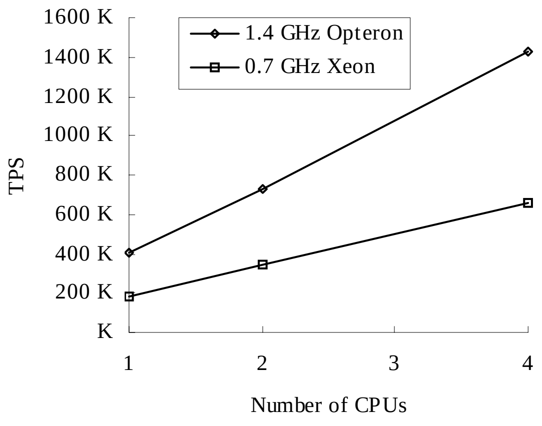


Figure 5. Search Scalability with Varying Number of CPUs

enabling and disabling on-disk write cache. The use of on-disk write cache is acceptable if the power supply to disk drives is securely backed up by UPS to cope with power failure.

Figure 3 shows that P*TIME can process 140K field update transactions per second (TPS) with four log disks used for parallel logging. For pure insert and delete workloads, P*TIME handles 80K and 100K TPS, respectively. The response time with on-disk write cache enabled (1~3 ms) is better than that disabled (9~16 ms). In both cases, the response time improves significantly with the reduced number of connections.

This figure also shows that the performance with on-disk write cache disabled converges to the performance with on-disk cache enabled as the number of connections increases. With the “deferred commit” option, P*TIME can deliver the same peak performance with one log disk with slightly better response time. However, with this

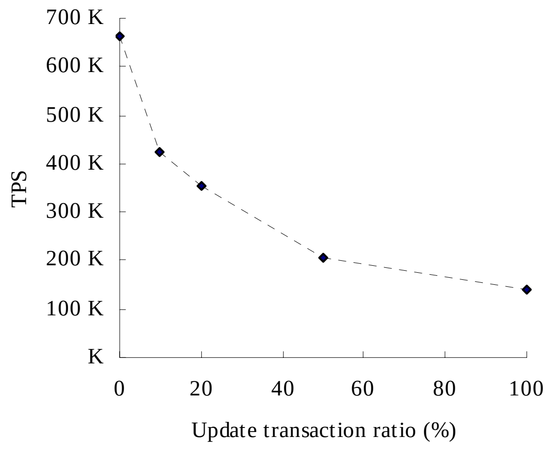


Figure 6. Mixed Workload Performance with Varying Ratio of Updates (WC disabled, 4800 connections)

option, the durability of updates is not guaranteed in case that the operating system failure occurs.

Figure 4 shows the scalability of P*TIME update performance with the varying number of log disks. As the number of log disks increases, the update throughput also increases until CPU, memory bus or IO channel capacity is saturated by a specific hardware configuration.

3.2. Search performance

Figure 5 shows that P*TIME can process 650K search transactions per second on the 4-way 700MHz Xeon server, and more than 1.4 million transactions per second on a 4-way 1.4GHz AMD Opteron server running RedHat AS 3.0 with 1MB L2 cache. Careful design of P*TIME internal data structures and algorithms minimizing L2 cache misses enables such linear search scalability with the number and the speed of CPUs.

Figure 6 shows the overall throughput when update and search transactions are intermixed, ranging from 100 % search to 100 % update. When the search/update ratio is 90/10, 80/20 and 50/50, the overall throughput is about 430K, 350K and 200K TPS, respectively.

3.3. Multi-tier performance

To evaluate P*TIME performance in a multi-tier application environment, we created a number of multithreaded Java application processes on several machines, which are connected to the P*TIME server process on the 4-way 700MHz Xeon server via gigabit Ethernet. In this environment, P*TIME shows 70,000 TPS for search-only workload and 42,000 TPS for the update-only workload. Transaction group shipping was turned on to utilize the server’s communication bandwidth efficiently. When only a single client thread is connected, P*TIME shows 5,800 search TPS with the average response time of 0.17ms or 1,320 update TPS with the

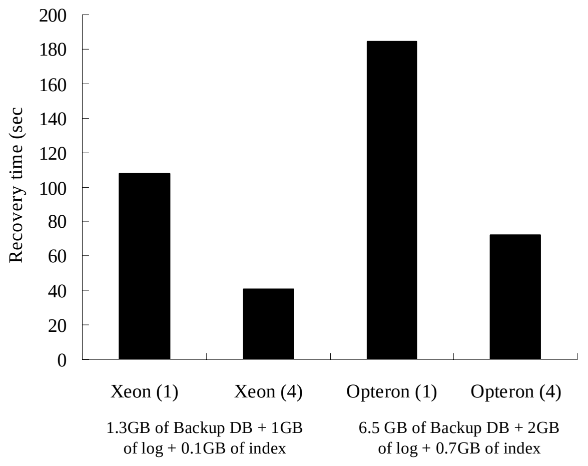


Figure 8. Database Recovery Time (The number in parentheses denotes the number of CPUs and disks.)

average response time of 0.76ms with WC enabled. Although these numbers represent significant drops from the peak engine performance numbers, they correspond to ten times in search and hundred times in update compared with a fully-memory-cached disk-centric database running in the same environment.

3.4. Restart time

To see the impact of P*TIME parallel recovery, we measured the time to recover the whole database from 1.3GB of backup database (8M records), 1GB of log records (10M update transactions), and 0.1GB of non-persistent index. The recovery time is broken down to loading the checkpointed backup database, replaying the log records, and rebuilding non-persistent indexes. The first column in Figure 8 shows that the total recovery time of the sequential recovery based on a single log disk and a single checkpoint backup database is 107 seconds, which is reduced to only 41 seconds by parallelizing all individual recovery steps: loading backup database, replaying log records, and rebuilding indexes. Note that even the sequential recovery time of P*TIME is shorter than that of existing in-memory RDBMS implementations.

To measure the restart time for a larger database, we scaled the database size five times (6.5GB of backup database and 0.7GB of non-persistent index for 40M records) and the log size twice (20M update transactions or 2 GB). Since this database size exceeds the process address limit of the 32 bit Xeon machine, we used the 4-way 1.4GHz 64 bit Opteron server with 16GB DDR memory and several 15000 rpm disks connected through U320 SCSI controller. The fourth column in Figure 8 shows that the recovery time with 4 CPUs and 4 disks is only 72 seconds, while the recovery time with 1 CPU and 1 disk is 184 seconds. This experimental result leads us to conclude that the database recovery performance of P*TIME is scalable with the number of log and checkpoint disks used and the CPU/IO capability of the underlying hardware system.

4. Stock Market Database Case

4.1. Challenges

The stock market database keeps track of the current state and history of individual stock item's bid-and-ask and settled price and volume data. It is critical for this database to minimize the latency in capturing the continuous stream of stock trading messages and

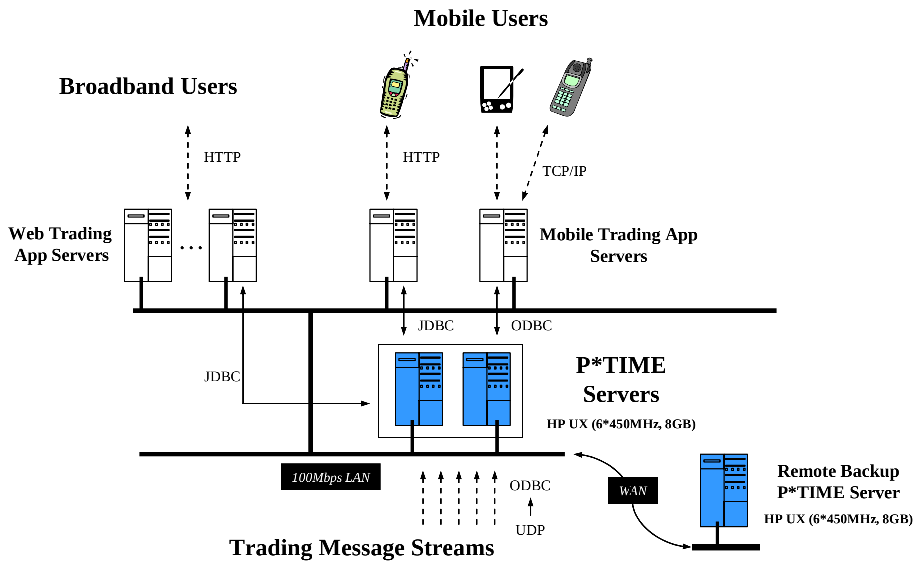


Figure 7. P*TIME-based Stock Market Database Servers

delivering updates to the interested users or responding to ad hoc user queries. Another important requirement is to lower the cost of quality service because of the competition among the stock brokerage firms. Conventional RDBMS implementations observing the ACID transaction quality is not adequate for cost-effective management of stock market data because of the amount of update volume to process.

4.2. Samsung Securities, Inc. case study

Samsung Securities, Inc. is the number 1 brokerage firm in Korea, serving 50K – 60K concurrent users out of 800K registered on-line users. Korean stock market is well known for its volatility and fast adoption of on-line trading because of the well-developed broadband and wireless infrastructures enabling easy access to the market database and the low transaction fee which resulted from the competition among many on-line brokerage firms.

Recognizing that implementing the market database servers with conventional RDMBS technology is cost-prohibitive, Samsung Securities, Inc. maintained a farm of about 50 C-ISAM-based market data servers communicating with the user's fat MS Windows client program in a custom protocol. Each server runs on a SUN Enterprise 3500 or 4500 hardware with six to eight CPUs. There is no clear separation of database and applications in this architecture, and the IT staffs has the burden of maintaining the application code base of manipulating C-ISAM and shared memory with ad hoc concurrency control.

When Samsung Securities, Inc. planned to launch the mobile trading service to mobile phone and PDA users in August 2002, it decided to separate the database server from the application because of the various difficulties that it has experienced in maintaining the C-ISAM-based server. Samsung Securities, Inc. chose P*TIME to manage the market database because its update scalability will lower the long-term capital expenditure and the standard RDBMS interface will lower the application maintenance cost.

Figure 7 shows the architecture of P*TIME-based market database server deployment at Samsung Securities, Inc. For the high availability reason, two copies of P*TIME servers are co-located with a single remote backup server. Since the trading message streams are broadcast via UDP from the stock exchange and the message processing burden is not severe for P*TIME, each P*TIME server is responsible for updating its own database. After a few months of operation, Samsung Securities, Inc. also decided to move the customer's profile of watch list groups from eight LDAP servers to the operational P*TIME servers. For this database, P*TIME servers are configured as active-active replicas. Although the operational systems were not planned to serve web trading services, the IT staffs of Samsung Securities, Inc. also implements the new functionalities to introduce in the web trading service using the operational P*TIME servers because of the ease of manipulating the database with SQL and JDBC.

4.3. Schema and Workload

The stock market database consists of 50+ relations. Figure 9 shows five representative relations. For each relation, the number of columns and the record length in bytes are shown as well as some column names. The primary key columns are printed in bold face.

STOCK_MASTER relation holds the reference information of stock items such as code, name, and trade status. The Korean stock market has 1700+ stock items managed by KSE and KOSDAQ. BIDNASK relation keeps track of ten closest bid and ask prices/volume pairs for each stock item. TRADE relation keeps track of the

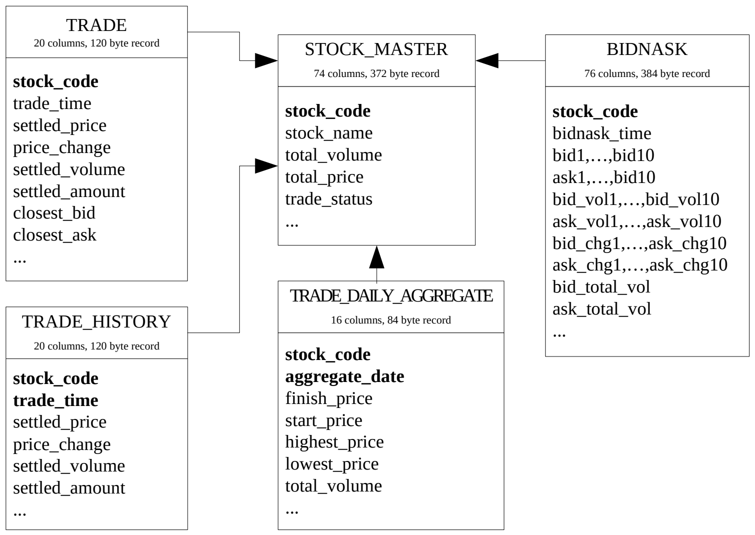
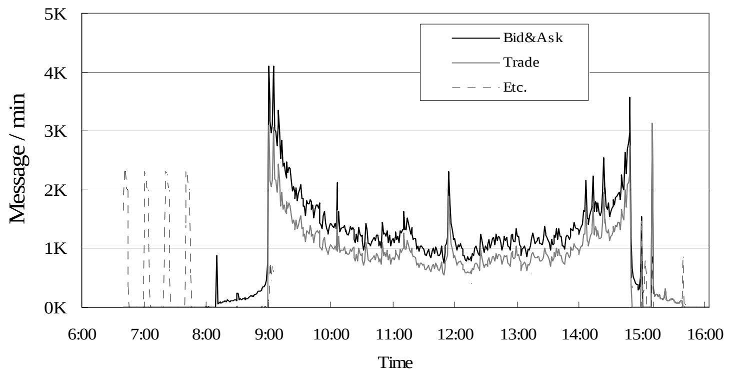
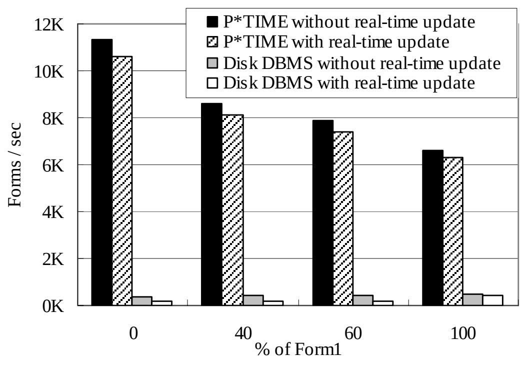


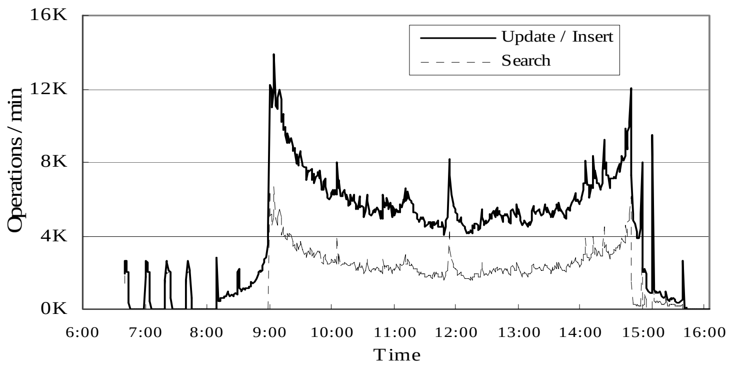
Figure 9. Representative Relations of P*TIME-Based Market Database



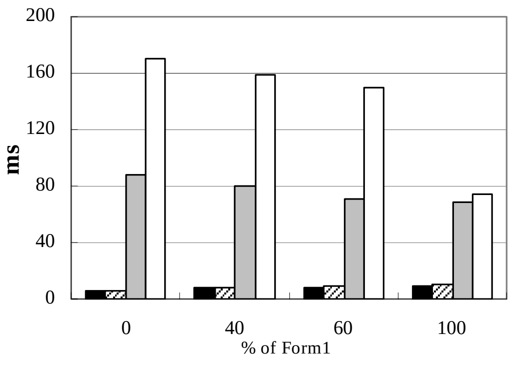
(a) Trading message distribution / minute



(c) Throughput



(b) Number of database operations / minute



(d) Response Time

QF1: **Retrieve the latest settled trade**

```
 SELECT    settled_price, price_change, settled_amount,
               settled_volume, closest_bid, closest_ask
 FROM      TRADE
 WHERE    stock_code=?
```

QF2: **Retrieve 60 day price changes**

```
SELECT     TOP 60 aggregate_date, start_price, finish_price,
                 highest_price, lowest_price, total_volume
 FROM        TRADE_DAILY_AGGREGATE
 WHERE      stock_code = ?
 ORDER BY aggregate_date DESC
```

(e) Form 1: Chart of 60 Day Price Changes of a Stock

QF3: **Retrieve the current price quote by joining three tables**

```
SELECT  <49 columns>
FROM     STOCK_MASTER M, TRADE T, BIDNASK B
WHERE   M.stock_code=? AND M.stock_code=T.stock_code
             AND M.stock_code=B.stock_code
```

QF4: **Retrieve the latest 20 settled trades**

```
SELECT      TOP 20 trade_time, settled_price, price_change,
                settled_volume, settled_amount
FROM         TRADE_HISTORY
WHERE       stock_code=?
ORDER BY trade_time DESC
```

(f) Form 2: Current Price Quote of a Stock

Figure 10. P*TIME-Based Stock Market Database Performance

latest settled trade information of each stock item, such as settled price and volume. In addition, it also keeps track of the closest bid and ask prices and volumes. TRADE_HISTORY relation stores the history of settled trades. TRADE_DAILY_AGGREGATE stores the daily aggregates of stock items such as high and low prices and traded volume. In addition, there are relations, which are not shown in the figure, for keeping track of options, futures, market indexes, and market statistics.

The challenging workload from DBMS perspective is processing the trading message stream from the stock exchange. Figure 10(a) shows the time distribution of one-day volume of real trading messages from Korean stock market. A major portion of this message stream consists of bid-and-asks and settled trades. Figure 10(b) shows the distribution of required update / insert and search queries to process the trading message stream.

For each bid-and-ask message, the following two updates are required.

- Update a row of BIDNASK with new values of bid and ask prices/volumes, etc.
- Update a row of TRADE with new values of the closest bid and ask information.

To process each settled trade message, the following operations are needed.

- Check the trade_status value of STOCK_MASTER.
- If the status is ok, search BIDNASK relation with a

given stock item to get the closest bid-and-ask match (bid1, ask1, bid_vol1, ask_vol1) because this information is not contained in the message.

- Update TRADE with new values of settled price, volume, etc.
- Insert the new trade record into TRADE_HISTORY.

Another type of major workload is query processing. Figure 10 (e) and (f) show two popular query forms. The first one displays the 60-day price changes of a stock item. The second one queries the current price quote of a stock item. Both forms generate two SQL queries in sequence, and each query is processed as a separate transaction.

In addition, there are periodic batch jobs conducted during night time or weekends to compute the daily, weekly, monthly aggregates.

4.4. Comparative P*TIME performance

To measure the comparative gain of using P*TIME over the existing DBMS implementation, we selected one of the easily accessible disk-centric DBMS implementations, and created two databases, one for P*TIME and another for the selected disk-centric DBMS, on the same Compaq ML 570 server that we use to report the experiment result of Section 3. In-memory DBMS implementations were not available for experimental comparison.

We used the real trading message stream of Figure 10 (a) to generate the update stream workload, and two query forms in Figure 10 (e), (f) to measure the concurrent query processing capability.

Figure 10 (c) and (d) shows the throughput and average response time with the varying ratio of two query forms. For each ratio, four values are shown. The first two values represent P*TIME and the remaining two values the disk-centric database. For each DBMS, the first value represents the performance without real-time market database update, and the second represents the performance with real-time market database update. The graphs shows that P*TIME is up to 40 times more scalable than the disk-centric database in throughput while the disk-centric database experiences severe degradation of response time.

4.5. Scalability with respect to stream data volume

To measure the scalability of update stream processing capability of P*TIME, we ran the experiment of accelerating trading message arrival using the one-day real trading message volume set. Figure 11 shows the throughput of concurrent query processing with the varying acceleration of trading message arrival rate. The query form 2, shown in Figure 10 (f), which has more interference with the real-time market database update is used to measure the throughput. In this figure, “No” means that there is no market database update load, 1x means the original speed of the stream, and 8x means that the message arrives at the eight times of the original speed.

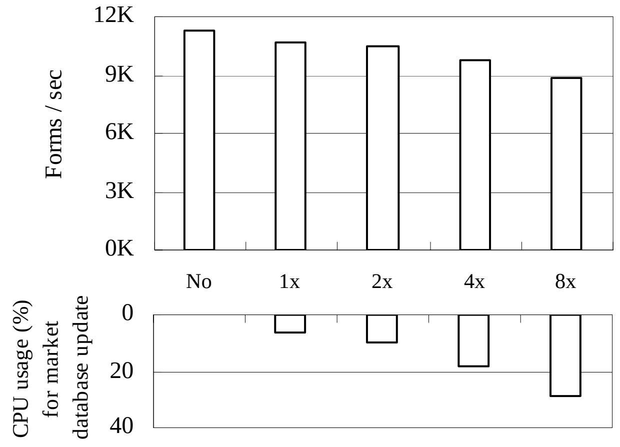


Figure 11. Concurrent Query Processing Throughput with Varying Rate of Trading Message Arrival

At 8x, about 2000 update transactions and 1000 search query transactions are performed per second to update the market database.

From Figure 11 we observe that the concurrent query processing is not much affected even though the rate of arrival changes from 1x to 8x. The minor drop in the concurrent query throughput is roughly proportional to the CPU usage for the real-time market database update.

5. Summary and Related Work

In this paper, we have presented:

- Rationale for the new OLTP DBMS for handling update-intensive stream workload which is frequently found in the so-called real-time enterprise applications.
- Architecture and performance scalability of P*TIME.
- Real-world deployment of P*TIME as the stock market database server.

With the exponentially growing gap between the CPU speed and the memory access speed, there has been much research lately on the L2-cache-conscious index structures ([3][4]), database layout ([6]), index concurrency control ([5]), and query processing ([7]).

Based on our experience of building, benchmarking, and deploying P*TIME in the real world transaction processing environment, we believe that careful implementation of L2-cache-conscious DBMS-internal protocols and algorithms such as the OLFIT and minimizing thread context switches leveraging the lightweight multithread architecture are more crucial than L2 cache-conscious data structures alone, which provide only marginal gain in improving overall throughput. As future work, we expect to formalize this experience further through experiment.

P*TIME architecture design and implementation has benefited from the author's experience of building and benchmarking the first-generation in-memory DBMS over several years [8][9][10], which was motivated by the exposure to the early in-memory query processing project at HP Laboratories [18].

Building P*TIME follows the vision of RISC-style DBMS [19]. Although we have not yet incorporated the self-tuning functionality, our compact code, which uses the C++ template feature extensively, coupled with the inherent simplicity of the memory-centric database performance model compared with the disk-centric one would be make it easier to incorporate the self-tuning capability.

As future work, we plan to extend P*TIME to incorporate the continuous query processing capability, and exploit the macro parallelism using a number of inexpensive SMP blades connected through a high-speed switch fabric. In another direction of research, we plan to refine the stock market database domain further so that it can serve as a benchmark database for the update-intensive stream workload.

Acknowledgement

The authors thank Gio Wiederhold at Stanford University for his invaluable comment on the early version of this paper. They also thank other members of P*TIME development team for their contribution.

REFERENCES

[1] Paul McJones, Ed.. The 1995 SQL Reunion: People, Projects, and Politics. Digital SRC Technical Note. 1997-018. http://www.mcjones.org/System_R/T

[2] Jack L. Lo, Luiz A. Barroso, Sujan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In Proceedings of the 25th Annual International Symposium on Computer Architecture, June 1998.

[3] Jun Rao and Kenneth Ross. Making B+-trees Cache Conscious in Main Memory. In Proceedings of ACM SIGMOD Conference, 2000.

[4] Kihong Kim, Sang K. Cha, and Keunjoo Kwon. Optimizing Multidimensional Index Trees for Main Memory Access. In Proceedings of ACM SIGMOD Conference, 2001.

[5] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In Proceedings of VLDB Conference, 2001. (Under patent application)

[6] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skousnakis. Weavering Relations for Cache Performance. In Proceedings of VLDB Conference, 2001.

[7] Peter Boncz, Stefan Manegold, and Martin Kersten. Data Architecture Optimized for the new Bottleneck: Memory Access. In Proceedings of VLDB Conference, 1999.

[8] Sang K. Cha, Jang Ho Park, Sung Jik Lee, Sae Hyeok Song, Byung Dae Park, S. J. Lee, S. Y. Park, and G. B. Kim. Object-Oriented Design of Main-Memory DBMS for Real-Time Applications. In Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications, 1995.

[9] Sang K. Cha, Jang Ho Park, Sung Jik Lee, Byung Dae Park, and J. S. Lee. An Extensible Architecture for Main-Memory Real-Time Storage Systems. In Proceedings of the 3rd International Workshop on Real-Time Computing Systems and Applications, 1996.

[10] Jang Ho Park, Yong Sik Kwon, Ki Hong Kim, Sangho Lee, Byoung Dae Park, and Sang K. Cha. Xmas: An Extensible Main-Memory Storage System for High-Performance Applications. Demo in Proceedings of ACM SIGMOD Conference, 1998.

[11] TimesTen Performance Software. http://www.timesten.com

[12] Juchang Lee, Kihong Kim, and Sang K. Cha. Differential Logging: A Commutative and Associative Logging Scheme for Highly Parallel Main Memory Databases. In Proceedings of IEEE ICDE Conference, 2001. (Under patent application)

[13] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In Proceedings of ACM PODS Conference, 2002.

[14] S. Krishnarmurthy, S. Chanrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Madden, F. Reiss, M. Shah. TelegraphCQ: An Architectural Status Report. IEEE Data Engineering Bulletin, Vol 26(1), March 2003

[15] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Transactions on Database Systems, Vol. 17, No. 1, pp. 94-162, 1992.

[16] IBM, ARIES Family of Locking and Recovery Algorithms. http://www.almaden.ibm.com/u/mohan/ARIES_Impact.html

[17] M-A. Neimat, TimesTen Caching Infrastructure and Tools, An Industry Session Presentation, In IEEE ICDE Conference, 2002.

[18] Tore Risch. The Translation of Object-Oriented Queries to Optimized Datalog Programs. HPL-DTD-91-9, Hewlett-Packard Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94303.

[19] Surajit Chaudhuri and Gerhard Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-style Database System. In Proceedings of VLDB Conference, 2000.