

FAME DATABASE  
PERFORMANCE  
BENCHMARK FOR  
TIME SERIES DATA

---

# TABLE OF CONTENTS

---

1	Introduction .....	1
2	Test Scenarios and Constraints.....	1
	Programming Languages .....	1
	Fame Local vs. Remote Database Access .....	1
	Datasets.....	1
	RDBMS Database Design: Schema and Index .....	2
	Physical Storage Caching .....	3
	Database Sizes .....	3
	Hardware and Software Specifications.....	4
	Other Constraints.....	5
3	Test Results .....	5
	Read Test 1 .....	6
	Read Test 2.....	6
	Write Test 1 .....	6
	Write Test 2 .....	7
4	Conclusion .....	8
5	Glossary .....	9

## INTRODUCTION

With industry efforts to reduce cost redundancies and inefficiencies, financial institutions, public sector organizations and other entities that rely on high volumes of economic and financial data every day need the best performance from their data management operations. Achieving the best price-performance for highly tuned and optimized data structures is imperative not only for saving costs, but also for helping to ensure faster time to market with new products and services and improving client satisfaction.

The goal of this benchmark study was to measure the performance of SunGard's Fame solution compared to three leading relational database management systems (RDBMS): Oracle, Sybase and MySQL. Several tests were designed specifically to focus on the performance of database engines when working with the unique and sophisticated properties of time series data – information collected at recurring intervals over a specific calendar period.

While the processing of time series data in an actual live environment involves writing, reading and validating data as well as complex analysis and computations, this study focuses purely on the write and read processes in order to demonstrate the raw performance of the databases. However, the write/read process is highly relevant on its own when time series data is integrated with third-party applications.

## TEST SCENARIOS AND CONSTRAINTS

In order to conduct a comprehensive and equitable evaluation of performance, the Fame performance benchmark uses different test scenarios and employs the optimal operating configurations of each RDBMS. The following describes the details of the test environment and scenarios, including the programming languages, data sets, and hardware and software configurations used, as well as the design, size and other details of the databases studied.

### ***Programming Languages***

The main programming languages used for the benchmark were Java and Fame Java Toolkit (a Java API formerly known as TimeIQ) for reading and writing to Fame databases. Java has mature and well-tested APIs for reading and writing to relational databases through Java Database Connectivity (JDBC). Using an industry standard programming language such as Java also helps ensure that the same data structures are used for reading and writing to the databases examined in this study.

### ***Fame Local vs. Remote Database Access***

In the test scenario, Fame databases were accessed locally rather than over a network. All databases were stored on the same machine as the tests were run in order to eliminate overhead from the networking interface serializing and de-serializing the data. This helps ensure that the speed of the network did not affect the performance of the databases and that a true test of database performance could be assessed. In addition, SunGard used a standard database configuration for Fame to reflect the manner in which the vast majority of Fame users typically update their Fame databases.

### ***Datasets***

Each database used for read tests utilized the entire FT North American Pricing data set for Open, Close, High, Low, Volume (O/H/L/C/V) prices for all securities from the point in which they started trading up to May 30, 2006. The data set contained approximately 10,800 securities and 28 million observations.

In order to closely represent a real-world scenario, not all securities have values over the same date ranges. For example, some securities are not trading anymore and therefore only have values up to a certain date. In addition, active securities that did not start trading at the same time will have different start dates. For the read tests, the databases read the dates in which the security actually traded. Since Fame is highly efficient in finding the first and last date with a time series value, these dates were calculated in Fame and exported to a table in the relational databases, which were then used to set the dates for queries that were eventually performed. This helped prevent the operation of finding the first and last date from affecting the tests.

It has been assumed that users or programs accessing the data will always request information for the time periods in which a security traded. If this is not the case, and data is requested outside of the date range for which the security actually has values with rows for those dates in the relational table, the Fame database uses a built-in calendar that recognizes that there are no values for all or part of the dates. Thus it will not try to retrieve values for those indices. In contrast, the RDBMS need to query each date and security combination to check if there are values, which causes significant degradation in relative performance.

For the write tests, the series was only updated for securities that were active on May 30, 2006, totaling 6,000 securities. By limiting the updates to these series, the test reflected a realistic environment by helping ensure that values were not added to series that have been inactive for long periods of time, which would create series with large segments of missing values.

#### ***RDBMS Database Design: Schema and Index***

The following is the schema that was used for the relational benchmarking pricing table. The index that was created on this table used the name and time\_stamp fields.

If an organization is storing time series data in an RDBMS, then most likely it will be searching for the data based on an instrument identifier or name and a start or end date. Therefore, it is natural to create an index on both the identifier name and the time stamp for each of the rows in the table.

FIELD	TYPE	NULL	KEY	DEFAULT	EXTRA
Time_stamp	datetime	Yes	Mul	Null	
Name	char (12)	Yes	Mul	Null	
Issue	char (12)	Yes		Null	
Open	float	Yes		Null	
High	float	Yes		Null	
Low	float	Yes		Null	
Close	float	Yes		Null	
Volume	float	Yes		Null	

For the purpose of the read tests, a number of different index methods were tested. Only the following were used because they produced the best performance for each RDBMS vendor. This was important to ascertain that the relational databases were tested in their optimized environments to help ensure a fair test.

**ORACLE example created on local Tablespace**

```

CREATE UNIQUE INDEX PRICING_IDX1
  ON PRICING(TIME_STAMP,NAME)
TABLESPACE FameDATA
NOLOGGING
PCTFREE 10
INITRANS 2
MAXTRANS 255
STORAGE(FREELISTS 1
  FREELIST GROUPS 1
  BUFFER_POOL DEFAULT)
NOPARALLEL
NOCOMPRESS

```

**SYBASE example created in local Sybase database on local device**

```

CREATE UNIQUE CLUSTERED INDEX pricing_idx
  ON pricing(name,time_stamp)
Go

```

**MySQL INNODB example created in database on local drive**

```

ALTER TABLE pricing_innodb
ADD UNIQUE pricing_innodb_idx(name,time_stamp);

```

**MySQL MYISAM example created on local device**

```

ALTER TABLE pricing_myisam
ADD UNIQUE pricing_myisam_idx(name,time_stamp) ;

```

**Physical Storage Caching**

Modern storage systems, such as redundant array of independent disks (RAID) or standalone storage systems, are capable of skewing input/output performance results because they tend to cache data adjacent to a specific seek operation. By randomizing the order of the securities that were queried, some but not all of the hardware caching optimizations may be negated. Therefore, it was challenging to gauge the effectiveness of this strategy as the caching algorithms and methods are not transparent. Ultimately, the reading of random objects was estimated to be a close approximation to the actual operation of live systems.

**Database Sizes**

It is important to note the sizes of the databases containing the same data. The physical size of each database, in megabytes, was as follows:

ORACLE	SYBASE	MY SQL-MY ISAM	MY SQL-INNOBDB	FAME
3491	4451	2897	3876	680

The sizes of the actual databases varied considerably. However, the Fame database was much smaller than the relational databases. This is due to the fact that Fame only stores the name and issue once for each time series object and not for each row, as is the case with relational databases. Fame also does not use the date-time index; it uses a signed integer instead.

This is significant when considering working with databases in memory because the size of the relational database will quickly become cost prohibitive due to the need for large machines to handle this memory.

Fame requires significantly less memory because it grows linearly. For this reason, the only conducted tests are those of memory-mapped Fame databases. SunGard acknowledges that running in-memory versions of the relational databases vendors would offer benefits similar to those presented for Fame.

### Hardware and Software Specifications

Two different test environments were used for the benchmark tests: a primary test environment and a secondary environment. The running of two test environments provided for more parallel testing as the test was developed, however the tests performed. The results of those tests, described below, are only presented for one environment.

PRIMARY TEST ENVIRONMENT	
Operating System	32 bit Linux Red Hat 3 EL
Hardware	Gateway E-4500D Server with 1x3.0Ghz processor, 2GB RAM, 1x160GB drive, Floppy, CD Rom, 16MB Graphics Card
Java	Java version 1.5.0_05
Fame	Release 9.2 Revision 2
Fame Java Toolkit (Fame Java API)	Version: 2.2 Chli Version 9.043 – Jchili Version: 2.2a Revision 0
MySQL Server	Version 5.0.17a – pro-gpl-cert-log
MySQL Client JDBC	mysql-connector-java-3.0.15-ga-bin.jar
Oracle Server	10.1.0.3.0–32 bit
Oracle Client JDBC	ojdbc14.jar, Oracle JD BC classes files will be named ojdcbbXX.jar, where XX is the Java version number
Sybase Server	Adaptive Server Enterprise 12.5.3
Sybase Client JDBC	jconn2.jar, jConnect™ for JDBC™/5.5 (Build 25578)/P/EBF12435/J DK13/Tue Feb 22 6:0 7:39 2005

SECONDARY TEST ENVIRONMENT	
Operating System	32 bit Linux Red Hat 2.4-21-37.ELsmp #1 SMP
Hardware	IBM x336, Dual Xeon 2.8Ghz, 8GB RAM, 2x146GB drives, running Linux, W2K3 Server, W2K Pro O/S's, RH EL 3
Java	Java version 1.5.0_05
Fame	Release 9.2 Revision 2
Fame Java Toolkit (Fame Java API)	Version: 2.2 Chli Version 9.043 – Jchili Version: 2.2a Revision 0
MySQL Server	Version 5.0.17a – pro-gpl-cert-log
MySQL Client JDBC	mysql-connector-java-3.0.15-ga-bin.jar
Oracle Server	10.2.0.1.0–64 bit
Oracle Client JDBC	ojdbc14.jar, Oracle JD BC classes files will be named ojdcbbXX.jar, where XX is the Java version number
Sybase Server	Adaptive Server Enterprise 12.5.3
Sybase Client JDBC	jconn2.jar, jConnect™ for JDBC™/5.5 (Build 25578)/P/EBF12435/J DK13/Tue Feb 22 6:0 7:39 2005

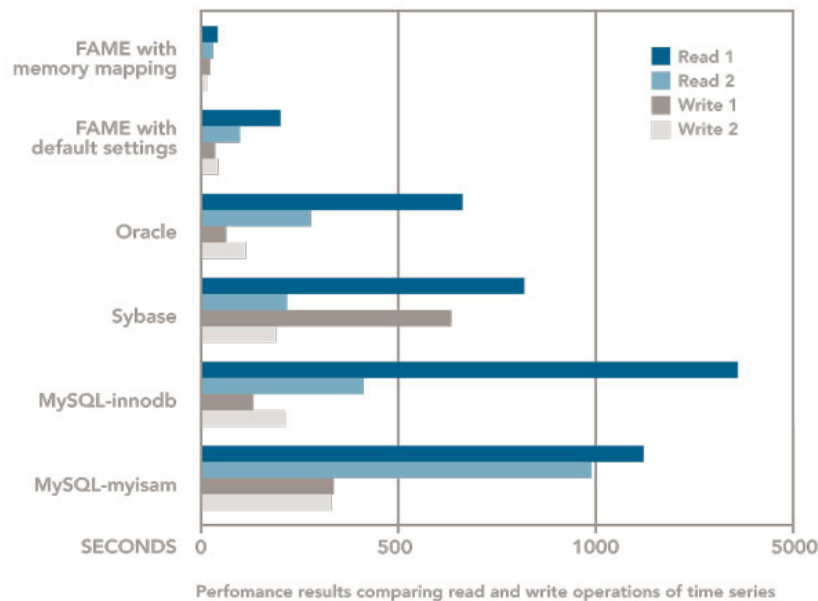
### Other Constraints

In addition, Fame's analytical capabilities were not used in the benchmark tests, and the databases created and used contained only time series data. However, performance results of the Fame analytical engine are presented at the end of this document for the benefit of existing Fame users and to demonstrate further performance advantages.

## TEST RESULTS

A number of read and write tests were performed on both test environments. Each test is described along with the resulting set of average times for each database, as well as a performance comparison between Fame using system defaults without custom settings and Fame using memory-mapped databases.

The test results show that Fame was significantly faster than relational databases and that the difference becomes larger, in absolute terms, the more observations that are working within each time series. For reading from databases, Fame was 2.5 to 11.5 times faster than RDBMSs when working with one year's worth of business data. Fame's read speed increased by 3.5 to seven times when working with the entire history of prices for stocks in FT North American pricing data set.



The same was true for writing to databases, with Fame yielding speeds of three to seven times faster for adding one day's worth of data. In addition, memory-mapping the Fame databases increased its speed by approximately 3.5 to 5.5 times over Fame with standard default settings.

Each test was repeated three times. The results presented in the following tables are the average times in seconds as well as in minutes:seconds.

**Read Test 1**

The first read test effectively read the entire database, one security at the time in random order. For each security, the Open, Close, High, Low, Volume time series were read for each day, from the first day to the last day that the security had a value.

RDBMS	SECONDS	MIN:SEC
Fame with memory mapping	33	:33
Fame with default settings	193	3:13
Oracle	654	10:54
Sybase	814	13:34
MySQL-innodb	1358	22:38
MySQL-myisam	1116	18:36

Memory mapping the Fame database made Fame's performance 5.79 times faster than using Fame with default settings. This means that Fame's performance could be further optimized if the memory mapped setting is enabled.

**Read Test 2**

Similar to the first read test, the second read test included the reading of only the last 260 days of pricing data. This scenario reflects a common practice by organizations for uses such as risk calculations.

RDBMS	SECONDS	MIN:SEC
Fame with memory mapping	24	:24
Fame with default settings	85	1:25
Oracle	213	3:33
Sybase	268	4:28
MySQL-innodb	395	6:35
MySQL-myisam	989	16:29

Memory mapping the Fame database made Fame's performance 3.60 times faster than using Fame with default settings.

**Write Test 1**

In the first write test, a new database was created and one year's worth of data was written to the database for each of the 6,000 securities, each one with values for Open, Close, High, Low, Volume. Two versions of this test were performed for Fame. In this test, changes were committed only after the updates were completed.

DATABASE & CONFIG	SECONDS	MIN:SEC
Fame and Fame Java Toolkit	11	:11
Fame and Fame Java Toolkit (commit 250)	23	:23
Oracle 16K batch post index	52	:52
MySQL-myisam 16K batch post index	122	2:02
MySQL-innodb 16K batch pre index	332	5:32
Sybase 16K batch post index	625	10:25

This write test was not performed using Fame memory mapped databases, as memory mapped databases are not recommended for situations with significant database growth. In this case, since the database was created from scratch, these operations should not be performed on memory mapped databases.

For the RDBMS tests, the data was committed in batches of 16,000 transactions. In three of the cases, creating the index after the data was loaded helped ensure that the relational databases operated more efficiently. Therefore, this method was used. The only exception was MySQL-innodb, which yielded better results when the index was created before the data was loaded. The goal was to ensure the RDBMSs were tested in their optimal states.

It is also worth noting that using the specialized bulk loading tools, rather than a Java application, for loading the data for the different RDBMSs yielded substantial improvements to performance. The largest improvement recorded was for Sybase-BCP, which loaded the data in 63 seconds compared to more than 10 minutes using a Java application.

### **Write Test 2**

The second write test used the same fully populated database as for the read tests. In this test, one observation (or day) was added to each series to reflect common practice, with testing performed with one commit every 100 series or rows. The updates in Fame were performed using ACCESS SHARED to allow users to read from the database at the same time as the updates were being performed.

This test repeated the update for 22 days and ensured that the hard drive was un-mounted between each test so that disk caching played no part in performance. This simulated a standard customer installation in which space is reclaimed and the database is compressed once every calendar month, leaving enough growth space for the next month's updates.

RDBMS	SECONDS	MIN:SEC
Fame and Fame Java Toolkit memory mapping	9	:09
Fame default settings and Fame Java Toolkit	31	:31
Oracle	97	1:37
Sybase	181	3:01
MySQL-innodb	205	3:25
MySQL-myisam	327	5:27

In this case, space was pre-allocated for the growth of the database to demonstrate the number for working with memory-mapped databases. Memory mapping the Fame database increased its performance by 3.29 times compared to using Fame with default settings.

## CONCLUSION

Time series data has special characteristics that require a fundamental design for assigning attributes to each time series. Fame makes intelligent use of these attributes when converting data from one frequency to another. When it comes to time series data, Fame's performance exceeds that of relational databases for several reasons.

Fame is specifically designed and optimized for rapid retrieval of time series data, employing a sophisticated indexing system, contiguous data storage, and dynamic caching intelligence to optimize the retrieval of time series data. Each time series is stored as a contiguous block, allowing all or part of the series to be scanned with a single disk read. Fame maintains data continuity by intelligently allocating disk space during the update procedure. Multi-level, read-write caching at the object, index and storage-block levels helps minimize disk access retrieval times.

This means that Fame's database size is smaller than that of RDMSs because it stores data more efficiently. Based on the test results, database size is congruent with performance. For example, for one issue name, Fame stores each name or object five times (once for each time series), whereas a relational database would need to store the object 2,600 times.

Within Fame, information is stored at its natural frequency, rather than forcing disparate data types to conform to a single structure or model. Rather than maintaining a date or time association in every database, Fame uses a true calendar that inter-relates every database point. It has an implicit understanding of the period of time, enabling it to recognize months of varying lengths, business days, holidays and leap years.

Each value is associated with the actual calendar date on which it was reported. In a relational database, weekly data would be captured into 52 periods per year, even when some years contain 53 observations. With Fame, comparisons between items with different frequencies are handled by a seamless conversion to any specified working frequency. Therefore, Fame does not have any relational requirements and it does not need to maintain a large index file.

By conducting this benchmark study, it is easy to appreciate how Fame's unique understanding of time series data is critical to the management and fundamental analysis of economic and financial data.

## FAME GLOSSARY

### **Fame**

A front-to-back market data solution, Fame provides investment professionals and intermediaries with a suite of products, including real-time market data feeds, Web-based desktop solutions, application hosting, data delivery components, tools for analytic modeling and a historical database for storing, managing, analyzing and delivering high-volume time series data. Currently used around the world by leading institutions in the financial, energy, and public sectors, Fame delivers complete data content, helping firms save time, minimize risk and enhance overall productivity.

### **Fame Software**

Fame Software provides a robust solution for storing and managing high-volume time series data – information collected at recurring intervals over a specific calendar period. Designed with a deep understanding of the unique and sophisticated properties of a time series, Fame is often considered the *de facto* standard for the storage and management of historical data. Used around the world by leading institutions in the financial, energy and public sectors, Fame is an established, reliable solution that quickly and easily integrates with downstream systems and custom applications.

### **Fame 4GL**

Fame 4GL is a proprietary, interactive scripting language that facilitates database access and analytics. It provides an environment for accessing, managing and manipulating data from Fame databases using a set of time-intelligent routines. Highly intuitive and easy to learn, Fame 4GL includes a full suite of time series statistical functional and forecasting procedures, giving users complete flexibility to define their own commands and functions.

### **Fame Java Toolkit**

Fame Java Toolkit (formerly known as TimeIQ) is a Java class library that facilitates the modeling and manipulation of Fame time series data in an object-oriented manner. It allows users to load data from the Fame databases into Java applications, presenting the full range of time series analytics in the form of “functors” or functions as separate classes. Fame Java Toolkit models time-series data storage, analysis and manipulation in an object-oriented manner, providing enterprise-level solutions for the storage and retrieval of historical data using Fame database technology. This allows developers to easily transform an underlying time series, such as timescale conversions, missing value detection, logical operators, statistical routines, and moving averages.

### **Fame C++ Toolkit**

Fame C++ Toolkit (formerly known as Fame C HLI) is an API that allows a C program to access a Fame database to retrieve time series data and convert it into native C structures. Time series data can be mapped into C vectors, allowing analysis in a C/C++ environment.

[www.sungard.com/fame](http://www.sungard.com/fame)

SunGard  
340 Madison Avenue  
7th Floor  
New York, NY 10173  
USA  
Tel: + 1-800-825-2518  
[www.sungard.com/dataintodecisions](http://www.sungard.com/dataintodecisions)

SunGard Data Systems, Inc. or its subsidiaries, including SunGard Data Management Solutions (collectively "SunGard"), is the exclusive owner of all copyrights and other intellectual property rights in this technical paper and other publications delivered electronically.

PLEASE NOTE THAT, EXCEPT AS EXPRESSLY SET FORTH IN AN EXECUTED LICENSE AGREEMENT BETWEEN YOU AND SUNGARD, ALL SUNGARD PRODUCTS ARE PROVIDED "AS IS" AND SUNGARD DISCLAIMS ALL WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. NOTHING IN THESE MATERIALS (OR IN ANY OTHER MATERIALS PROVIDED TO YOU) CONSTITUTES A REPRESENTATION OR WARRANTY REGARDING SUCH PRODUCTS.

No parts of this document may be reproduced, transmitted or stored electronically without SunGard's prior written permission. This document contains SunGard's confidential or proprietary information. By accepting this document, you agree that: (A) if a pre-existing contract containing disclosure and use restrictions exists between your company and SunGard, you and your company will use this information subject to the terms of the pre-existing contract; or (B) if no such pre-existing contract exists, you and your Company agree (1) to protect this information and not reproduce or disclose the information in any way; and (2) that SunGard makes no warranties, express or implied, in this document, and SunGard shall not be liable for damages of any kind arising out of use of this document.

DISCLAIMER: The screens and illustrations are representative of those created by the software, and are not always exact copies of what appears on the computer monitor. The material in this document is for information only, and is subject to change without notice. SunGard reserves the right to make changes in the product design and installation software without reservation and without notice to users.

**©2009 SunGard.**

**Trademark Information: SunGard, the SunGard logo, Fame and all Fame products are trademarks or registered trademarks of SunGard Data Systems Inc. or its subsidiaries in the U.S. and other countries. All other trade names are trademarks or registered trademarks of their respective holders.**