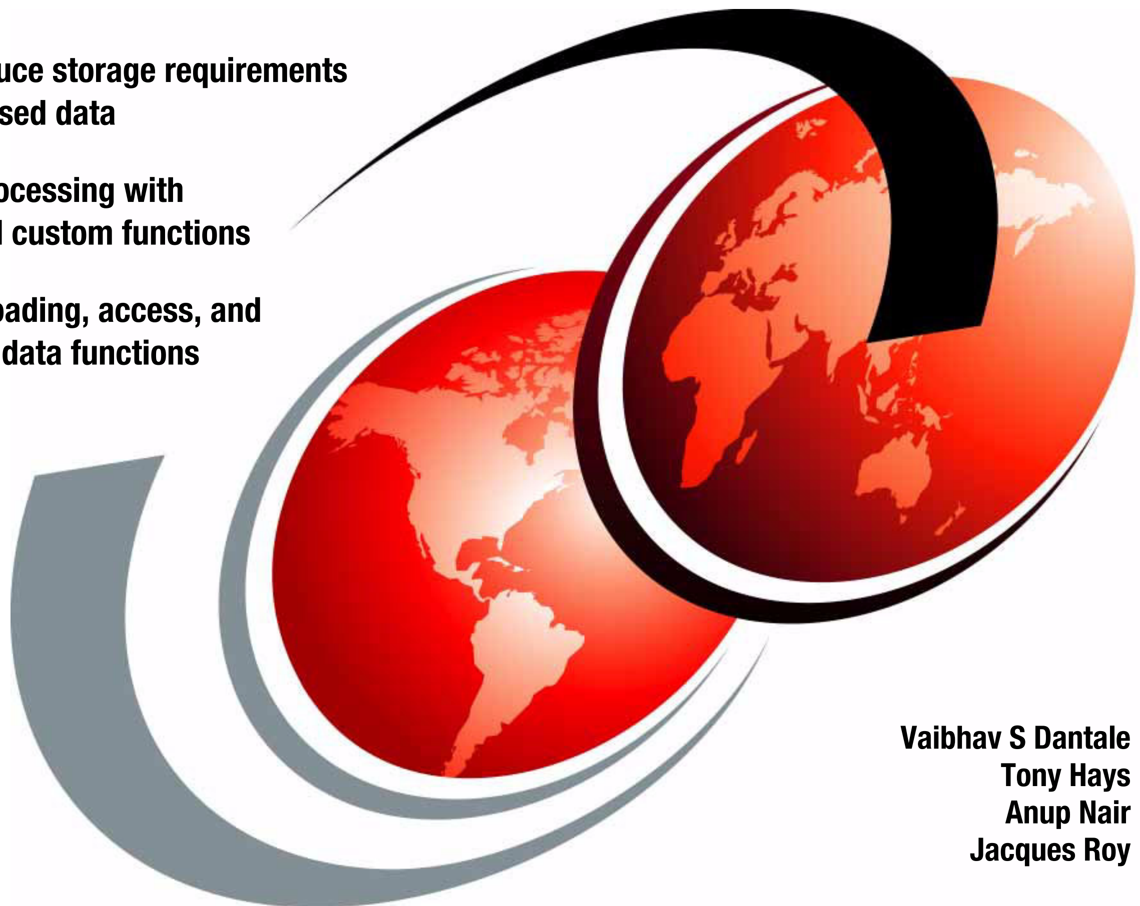
IBM® Information Management Software

Solving Business Problems with Informix TimeSeries

Greatly reduce storage requirements for time-based data

Simplify processing with built-in and custom functions

Speed up loading, access, and retrieval of data functions



Vaibhav S Dantale
Tony Hays
Anup Nair
Jacques Roy

Redbooks

ibm.com/redbooks

International Technical Support Organization

Solving Business Problems with Informix TimeSeries

September 2012

SG24-8021-00

Note: Before using this information and the product it supports, read the information in "Notices" on page vii.

First Edition (September 2012)

This edition applies to Informix Version 11.70, fixpack 5

© Copyright International Business Machines Corporation 2012. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices vii
Trademarks viii

Preface ix
The team who wrote this book x
Acknowledgement xi
Now you can become a published author, too! xii
Comments welcome xii
Stay connected to IBM Redbooks publications xiii

Chapter 1. Overview of IBM Informix 1
1.1 Building a smarter planet with Informix 2
1.2 Time series data 5
1.3 Benefits over relational model of storing data 5
1.4 The Informix TimeSeries approach 6
1.5 Informix TimeSeries components 7
1.5.1 TimeSeries ROW element sub-type 7
1.5.2 TimeSeries “out of row” storage containers 8
1.5.3 Types of time series data 9
1.5.4 TimeSeries calendars and calendar patterns 9
1.5.5 Informix TimeSeries system tables 11
1.5.6 SQL routines and APIs 13
1.5.7 Virtual table interface 13

Chapter 2. A use case for the Informix TimeSeries feature 15
2.1 Customer overview 16
2.2 The initial data model 17
2.3 Applying a time series data type to the relational data model 21
2.4 Final considerations 23

Chapter 3. Defining your TimeSeries environment 25
3.1 Schema 26
3.2 Relational and time series data working together 28
3.3 Regular or irregular data 29
3.4 The TimeSeries element 31
3.5 Calendars 33
3.6 Time zone issues 34
3.7 Loading data 34
3.8 Data cleansing and corrections 35

© Copyright IBM Corp. 2012. All rights reserved.

3.9 High availability 35
3.10 Backup 35
3.11 Purging data 36
3.12 Spatial requirements 36

Chapter 4. Implementing Informix TimeSeries 37
4.1 Schema definitions 38
4.2 Space calculations 38
4.2.1 Space calculations for relational storage 38
4.2.2 Space calculations for indexes 41
4.2.3 Space calculations for Informix TimeSeries storage 43
4.2.4 Container calculations for Informix TimeSeries storage 45
4.2.5 Time series storage needs versus relational storage needs 45
4.3 Loading time series data 46
4.3.1 Small amounts of data 46
4.3.2 Large amounts of data 47

Chapter 5. Querying TimeSeries data 53
5.1 Basic relational views 54
5.1.1 Expression-based relational views 55
5.1.2 When to use relational views 56
5.2 Using the TimeSeries SQL API 57
5.2.1 TimeSeries SQL API functions 57
5.2.2 When to use TimeSeries SQL functions 60
5.3 Custom functions. 60
5.3.1 Creating a custom function 61
5.3.2 When to use custom functions 63
5.4 Query examples 64
5.4.1 Getting a day of data for a customer. 64
5.4.2 Getting a day of aggregated data for a customer 65
5.4.3 Comparing two different days for a customer 65
5.4.4 Get peak usage for a time period, population subset 66
5.4.5 Billing for a subset of customers 69

Chapter 6. Managing the ecosystem 75
6.1 System management and monitoring 76
6.1.1 Monitor TimeSeries containers using API 76
6.1.2 Querying the TSContainerTable system table 79
6.1.3 Monitoring with IBM Informix OpenAdmin Tool. 80
6.1.4 Managing with OAT and APIs. 87
6.2 Performance considerations for TimeSeries data 95
6.2.1 Storage consideration 96
6.2.2 Data distribution statistics 96
6.2.3 Memory consideration. 97

6.2.4 Access consideration 97
6.3 Availability of data 97
6.4 Interoperability and the complete ecosystem 98
6.4.1 Virtual table interface 99
6.4.2 TimeSeries APIs 99
6.4.3 TimeSeries APIs in Informix stored procedure 100

Appendix A. Reference material 101
A.1 Online documentation 102
A.2 The IBM developerWorks wiki 102
A.3 PDF manuals 102

Appendix B. Enterprise historian database example 105
B.1 Wind power generation historian 106
B.2 Disk space savings 106
Standard RDBMS approach 106
Informix TimeSeries data type approach 107
B.3 Application development and performance 109
B.4 Interoperability 115
B.5 Summary 115

Appendix C. Distribution grid monitoring enabler 117
C.1 Solution overview 118
C.2 Business benefits 119
C.3 Challenges 120
C.4 Solution 121
C.5 Performance 121
C.6 Ease of development 123
C.7 Summary 126

Related publications 127
IBM Redbooks publications 127
Other publications 127
Online resources 128
Help from IBM 128

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

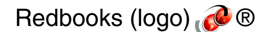
This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

© Copyright IBM Corp. 2012. All rights reserved.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®
Cognos®
DataBlade®
developerWorks®
IBM®
Informix®
InfoSphere®
Optim™
Redbooks®
Redbooks (logo)  ®
WebSphere®

The following terms are trademarks of other companies:

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.

Preface

The world is becoming more and more instrumented, interconnected, and intelligent in what IBM® terms a *smarter planet*, with more and more data being collected for analysis. In trade magazines, this trend is called *big data*.

As part of this trend, the following types of time-based information are collected:

- Large data centers support a corporation or provide cloud services. These data centers need to collect temperature, humidity, and other types of information over time to optimize energy usage.
- Utility meters (referred to as *smart meters*) allow utility companies to collect information over a wireless network and to collect more data than ever before.

IBM Informix® TimeSeries is optimized for the processing of time-based data and can provide the following benefits:

- Storage savings: Storage can be optimized when you know the characteristics of your time-based data. Informix TimeSeries often uses one third of the storage space that is required by a standard relational database.
- Query performance: Informix TimeSeries takes into consideration the type of data to optimize its organization on disk and eliminates the need for some large indexes and additional sorting. For these reasons and more, some queries can easily have an order of magnitude performance improvement compared to standard relational.
- Simpler queries: Informix TimeSeries includes a large set of specialized functions that allow you to better express the processing that you want to execute. It even provides a toolkit so that you can add proprietary algoritms to the library.

This IBM Redbooks® publication is for people who want to implement a solution that revolves around time-based data. It gives you the information that you need to get started and be productive with Informix TimeSeries.

© Copyright IBM Corp. 2012. All rights reserved.

The team who wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Vaibhav S Dantale is an IT Advisory Engineer in the IBM Information Management Division, in Pune, India. After joining IBM Informix in 2004, he worked on several IDS components.

Recently, his primary focus has been on IDS TimeSeries features. He is a key player in migrating Distribution Grid Monitoring Enabler on Informix and paving the way for Informix in the distribution sector. Similarly, he also has a major role in positioning Informix TimeSeries in space for historian solutions for power generation and manufacturing industries. Vaibhav holds the Bachelor's degree in Computer Technology from Nagpur University, India.

Tony Hays is a Senior IT Architect in the Enterprise Integration Communications practice in the United States. He has more than 25 years of IT experience. He has worked for IBM for 16 years and is certified in Integration Architecture. His areas of expertise include IBM WebSphere® MQ, WebSphere Message Broker, and Informix TimeSeries.

Anup Nair is a senior member of the Informix Competitive Technologies and Enablement team based in the US. He has a Bachelor's degree in computer engineering and a Master's degree in Computer Science Anup has more than 21 years of industry experience and he has held positions within management, marketing, software development, and technical support teams. He has authored various publications including Redbooks documents and technical white papers. He teaches classes and presents at conferences, user group meetings, and seminars worldwide related to Informix.

Jacques Roy is well-known in the Informix community as an author, speaker, and active blogger. Jacques is the manager and architect of the application development services and extensibility group of the Informix lab. This included the development of the TimeSeries product. He is the author of *IDS.2000: Server-Side Programming in C* and the lead author of *Open-Source Components for IDS 9.x*. He is also the author of multiple technical IBM developerWorks® articles on a variety of subjects, and co-author of several IBM Redbooks publications. He is a frequent speakers at data management conferences, the International Informix Users Group (IIUG) conference, and users group meetings.

Acknowledgement

Thanks to the following people for their contributions to this project:

Abhay Patra is Lead Architect for DGM asset and working as an Application Architect in GBSC Energy and Utility Industry team, based out of Pune. He won an IBM GBS Global Technical Achievement Award in 2010 and has filed a patent on Smart Grid device commissioning. Abhay is playing a significant role as a solution architect in most of the Smart Grid opportunities in India.

Inge Halilovic is a technical lead in Informix Information Development, based in San-Francisco.

We would also like to thank the following members of the Informix development team for their help clarifying technical questions:

- Kevin Brown
- Mark Ashworth
- Simon David (Cosmo)
- Jeff McMahon
- Lance Feagan

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks publications

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

1

Overview of IBM Informix

IBM Informix combines the robustness, high performance, availability, and scalability needed by today's modern business. This chapter provides a brief overview of Informix and introduces the Informix TimeSeries approach and components.

© Copyright IBM Corp. 2012. All rights reserved.

1.1 Building a smarter planet with Informix

Complex, mission-critical database management applications typically require a combination of online transaction processing (OLTP), batch, and decision-support operations, including online analytical processing (OLAP). Meeting these needs is contingent upon a database server that can scale in performance as well as in functionality. The database server must adjust dynamically as requirements change from accommodating larger amounts of data, to changes in query operations, to increasing numbers of concurrent users. The technology should be designed to efficiently use all the capabilities of the existing hardware and software configuration, including single and multiprocessor architectures. Finally, the database server must satisfy user demands for more complex application support, which often uses nontraditional or "rich" data types that cannot be stored in simple character or numeric form.

Keeping its strong philosophy of innovation, every newer version of Informix continues the tradition of constantly evolving and enhancing its industry leading database technology: powerful, easy to use, easy to manage, high quality, and high performance for OLTP and mixed OLTP/OLAP environments. Informix allows businesses of all sizes and purposes to resolve complex business challenges in a simplified, efficient, and low cost IT environment. As with previous versions, Informix also continues to enhance and strengthen its functionality in various areas of the database server, including administration, embeddability, availability, application development, supportability, and security.

Informix is built on Dynamic Scalable Architecture (DSA). It provides one of the most effective and efficient solutions available—a next-generation parallel database architecture that delivers mainframe-caliber scalability, manageability, and performance; minimal operating system overhead; automatic distribution of workload; and the capability to extend the server to handle new types of data. There are numerous articles written about the components and workings of DSA.

For brevity and simplicity, Figure 1-1 illustrates the “big picture” of the Informix DSA.

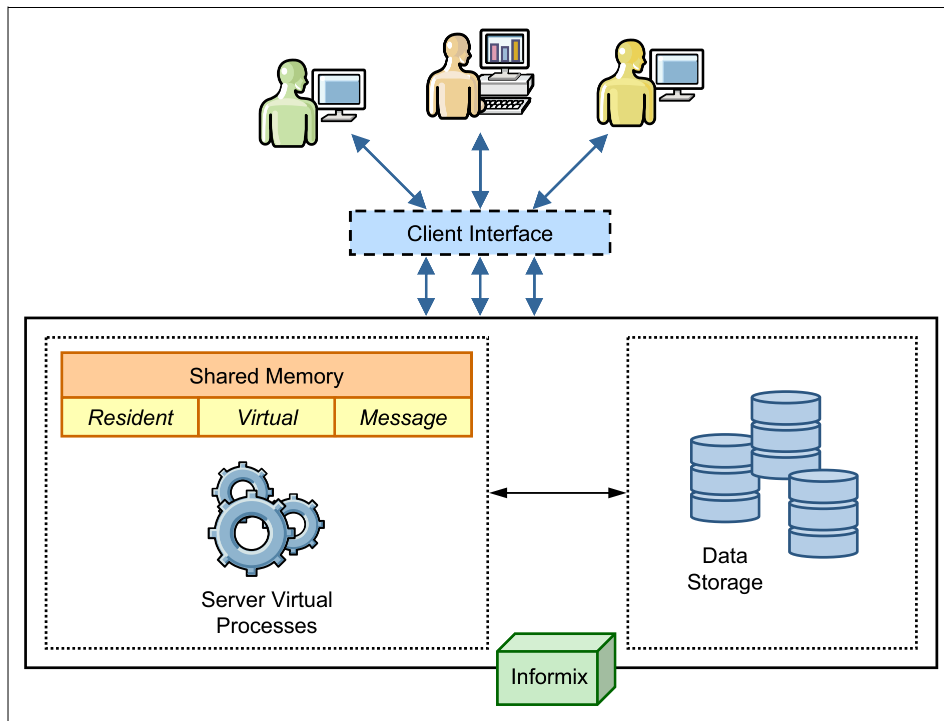


Figure 1-1 Informix DSA, the big picture

One of the biggest advantages of having the DSA is the ability of the server to extend its capability by integrating customized solution extensions based on complex data types. Informix delivers proven technology that efficiently integrates new and complex data directly into the database. It handles time series, spatial, geodetic, Extensible Markup Language (XML), video, image, and other user-defined data side by side with traditional existing data to meet today’s most rigorous data and business demands.

Informix has a built-in API extension layer, known as a *datablade*, which helps application extensions to seamlessly integrate with the core server functionality. Figure 1-2 illustrates the integration of the Informix extensibility layer within the server.

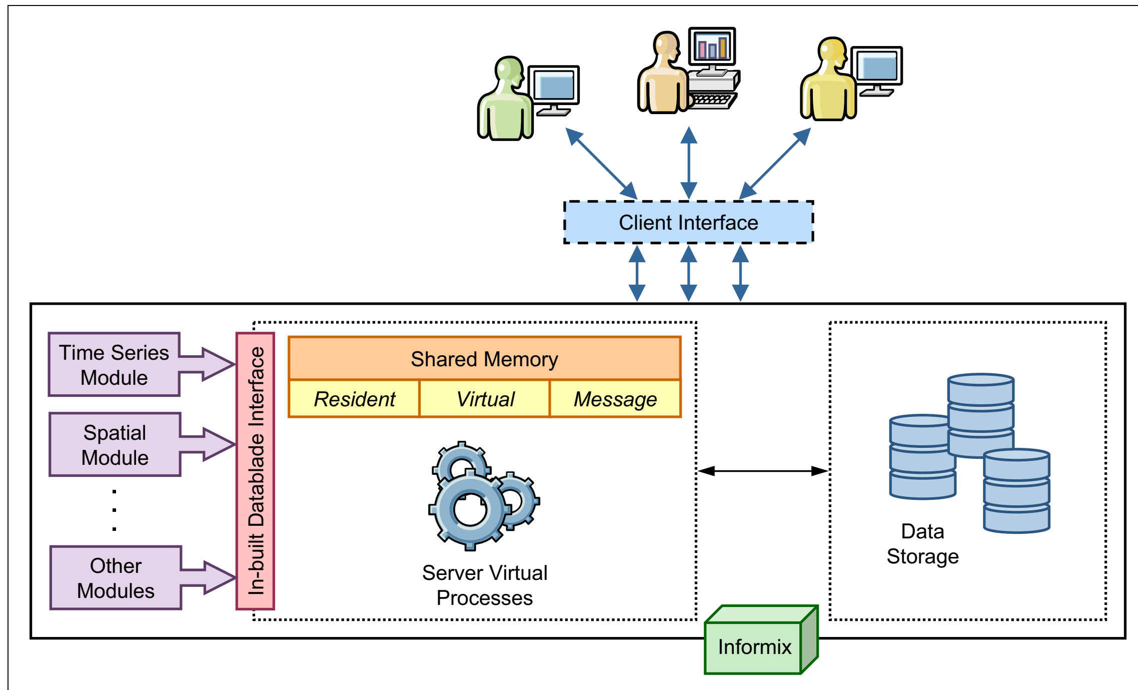


Figure 1-2 Extending Informix Server

To further strengthen support for the IBM smarter planet initiative, Informix continues its focus on enhancing its usability for building real-life solutions based on sensors and other instrumentation components that are used extensively in the Energy and Utilities (E&U) industry for the Meter Data Management (MDM) solutions for smart meters. MDM is one of the cornerstones of smarter energy solutions, delivering benefits throughout the entire value chain, from utility suppliers and operators to consumers. Time stamped data is a key component in smart metering solutions. The industry leading built-in Informix TimeSeries data type technology helps manage this time stamped (time series) data with extreme performance, high storage saving, and simplified usability.

1.2 Time series data

Time series data is a sequence of data points, measured typically at successive times spaced at uniform time intervals. Basically, it is a set of data where each reading element or row is time stamped.

Depending on the frequency of data acquisition through sensors and other data retrieval mechanisms, the volume of time stamped data can be massive; when stored in tall-thin relational tables, this data can require extremely large amounts of storage space. The storage requirement increases further when the table that stores the data mandates building multiple indexes on the time series and other data element columns to satisfy application and data retrieval requirements. Managing and retrieving such a huge volume of data can require complex querying, resulting in high disk I/O and reduction in performance.

The Informix TimeSeries data type alleviates the classical relational time series implementation issues of storage, performance, and complex querying. The Informix TimeSeries data type provides a native Object-Relational data implementation that requires less storage, while reducing I/O and increasing performance. Specialized APIs can alleviate the complexity of querying.

Customer case studies and internal benchmarks have revealed that using the TimeSeries data type yields 50% - 70% storage savings, 20 - 30 times performance improvement in data loading, and over 70 - 90 times performance increase in report processing compared to a typical relational implementation.

1.3 Benefits over relational model of storing data

The traditional relational approach stores one row each for every time series entry, thus storing redundant information like device identifiers multiple times. For faster retrieval of information, a composite index must be created on this identifier column and time stamp. If we consider the example of smart meter solutions, say an historical or Supervisory Control And Data Acquisition (SCADA) system, where the system needs to collect meter readings with certain fixed frequency, the single location ID/Meter ID (meter_id) is stored as many times as the time stamp of the event.

As mentioned previously, you create a composite index on meter_id and time stamp (Table 1-1), but this eventually yields a massive amount of data, which in turn brings in administration overhead, performance challenges, and hardware investment. These challenges often become major roadblocks for further enhancement in solutions and services.

Table 1-1 Representation of data stored in traditional RDBMS approach

Meter_id	Timestamp	Current	Voltage	Resistance
1	2010-12-01 01:00:00.0000	4.0	160	40
1	2010-12-01 01:00:10.0000	4.5	155	35
1	2010-12-01 01:00:20.0000	5.0	165	33
..	..	..	..	..
2	2010-12-01 01:00:00.0000	5.0	175	44
2	2010-12-01 01:00:10.0000	4.5	160	35
..	..	..	..	..

Due to these challenges, the data reading collection frequency has to be kept low, say every half hour, which for a 24 hours cycle is 48 data points for a single identifier (2 data points per hour times 24). Increasing the frequency of the data reads to every 15 minutes causes a massive increase in data (4 data points per hour times 24). This increase is mainly because the storage requirement immediately doubles because the reading fetches 96 data points instead of the earlier 48 data points in a 24 hour period for a single identifier. For this reason, it is often challenging to implement a near real-time solution in a relational model that yields both storage saving and performance benefit, without special tweaking.

Another major challenge is to perform time-based analytical operations on relational data. Most of the time, the application developer ends up developing proprietary and complicated program logic to perform time-based analysis.

1.4 The Informix TimeSeries approach

To tackle the issues in the relational model, Informix introduced a different perspective for looking at time series data, implementing the *native* TimeSeries data type. TimeSeries creates a single row for each time-stamped tag or tick identifier that says *meter_id* and simply appends subsequent readings from the same meter to that row, as shown in Table 1-2. The data is physically ordered on disk. Thus, the number of rows in a TimeSeries table will be the number of meters that you are monitoring. The index is also usually created only on the key (meter_id) column and therefore does not require the time stamp to be a part of the index. Thus, the storage requirement compared to a traditional relational approach is reduced by 50% - 80%.

Additionally, because the index size is small and the data is physically ordered on the disk, querying data requires much less I/O than for a relational approach. Performance using the Informix TimeSeries approach is better for both loading and querying data.

Table 1-2 Representation of data stored in Informix TimeSeries column

Meter_id	TimeSeries column
1	[(4,160,40), (4.5,155,35), (5,165,33)….]
2	[(5,175,44), (4.5,160,35). .]
. .	. .

1.5 Informix TimeSeries components

Before delving into the details of TimeSeries functions, it is of utmost importance to understand the terms and components of the TimeSeries solution as described in the following sections.

1.5.1 TimeSeries ROW element sub-type

A TimeSeries *element* is a set of data for a particular time stamp. What it basically means is that for a specific key value and a particular time stamp for that key value, all the associated information or data for that time stamp is stored in one object as a column inside that row. There can be multiple pieces of information with various time stamps for this key value, all stored in the corresponding column in that row. Internally, the TimeSeries data is stored inside a table column as a special ROW object. Metaphorically speaking, a TimeSeries row inside a table can be considered as a *table-inside-a-table*.

You can have multiple TimeSeries fields and columns inside a table. Each of these TimeSeries fields and columns can have a different element definition based on the business needs. The table-inside-a-table concept is implemented by the use of an Informix extended data type called ROW. While defining this subtype for TimeSeries, it is mandatory to define the first field or column as a unique DATETIME YEAR TO FRACTION(5), as shown in Example 1-1.

Example 1-1 Creating a row type to hold TimeSeries data

```
CREATE ROW TYPE LocReading (
   tstampDATETIME YEAR TO FRACTION(5), -- Mandatory
   highFLOAT,
   lowFLOAT,
```

```
   closeFLOAT,
   volFLOAT
);
```

1.5.2 TimeSeries "out of row" storage containers

As indicated earlier, a time series is a special column in a table. Internally this column can include a header and a set of elements associated with the time series. Currently a table row has a size limit of 32 KB. Because the TimeSeries data for a key is stored in the same row, it is likely that for a large time series, the incoming data would exceed the 32 KB limit. To address this issue, the Informix TimeSeries feature has implemented the concept of storing data "out of row" in external storage space, but within Informix database logical storage objects (dbspace). These "row external" storage spaces used for holding the Informix TimeSeries data are known as *containers*.

Containers are specialized logical storage spaces that are created inside a dbspace (Figure 1-3). You can store up to 64 GB (for a 4 KB page size) of data in one TimeSeries container. You can store multiple time series in a single container. You can have multiple containers in a dbspace.

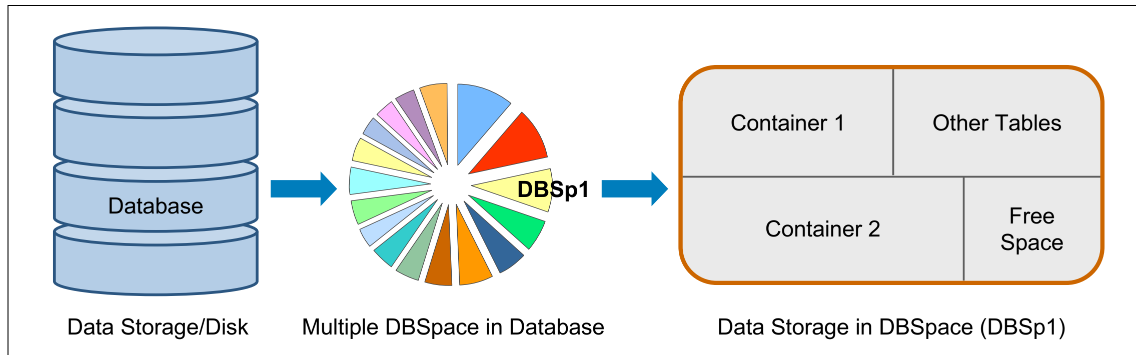


Figure 1-3 Containers in database storage

Because a container exists inside a dbspace, we need to use specialized routines to create containers in a dbspace before they can be used for storage. The TSContainerCreate() function helps you create this storage:

```
TSContainerCreate(container_nameVARCHAR(18),
   dbspace_nameVARCHAR(18),
   ts_row_typeVARCHAR(18),
   container_size INTEGER,
   container_growthINTEGER);
```

Example 1-2 illustrates creation of the container named `taqtrade_day` in the dbspace `dbspace1` with an intial size of 1 MB and a growth size of 1 MB. This container stores the data for the TimeSeries ROW type `taqtradeday_t`.

Example 1-2 Sample use of TSContainerCreate

```
EXECUTE PROCEDURE TSContainerCreate('taqtrade_day', 'dbspace1',
'taqtradeday_t', 1024, 1024);
```

1.5.3 Types of time series data

Time series data includes both *regular* and *irregular* types of data.

Regular

A regular time series stores data for regularly spaced time intervals. In a regular time series, each interval between elements is the same length, hence they are predictable.

A regular time series uses the concept of *offset*, referring to a mapping between the time point associated with an element and its position relative to the start of the time series. Individual time stamps are not stored in regular time series; instead, they are computed from the element's offset. Not storing time stamps for millions of records results in storage savings.

A regular time series is appropriate for applications that record entries at predictable points in time. Smart metering is one of the examples where we store the information with a certain fixed frequency, such as electricity power usage data that is recorded by smart meters at regular interval of 15, 30 minutes, and so on.

Irregular

An irregular time series manages information for arbitrary points in time. An irregular time series is appropriate when the data arrives unpredictably, such as when the application records every stock trade or when electricity meters record random events such as low battery warnings or low voltage indicators.

1.5.4 TimeSeries calendars and calendar patterns

The Informix TimeSeries feature uses calendars to define the context of the time stamped data. The calendar definition includes the first valid time and when and how often data is collected. For example, you can specify that data is collected every hour during business hours on weekdays and is not collected on Saturday and Sunday. You can define your own calendar or use a predefined calendar.

A calendar includes the following key components:

- Calendar Start Date: The date when the data can be stored.
- Calendar Pattern: This determines the granularity of the data and the time (interval) when the data is going to arrive. This granularity can be of any magnitude - year, month, week, day, hour, minute, or second. The occurrence of data is defined using a repeating pattern of “on” and “off” intervals. Data is only stored in “on” intervals, for example, every day, every other month, or work hours.
- Interval: The elapsed time between data points, this value can be qualified in seconds, minutes, hours, days, weeks, months, or years.
- Pattern Start Date: It is the first point in time at which the calendar pattern is applied. The pattern start date can be different than the calendar start date, but if different, it must be later than the calendar start date.
- Offset for regular TimeSeries: Because the occurrence of the next time series in a regular time series is predictable (calculated from calendar start date and pattern), the time stamp is not stored. Instead, only the offset to the first TimeSeries data is stored.

The information from these calendar components is recorded in the TimeSeries header that is stored in the beginning of the row cell for which the column type is defined as a time series.

The following information is also stored in the header of the TimeSeries row cell which sets its context of the TimeSeries to the data:

- Calendar: Time period for which data is found. This also includes the calendar patterns described previously.
- Origin: Time origin of the time series.
- Threshold: In-row storage threshold that defines whether the data is stored in-row or in a container.
- Container: Where to store the out-of-row data. It contains information about where data is stored in a dbspace.
- Metadata: Optional data added by the time series creator.

1.5.5 Informix TimeSeries system tables

Informix stores information about calendars, patterns, containers, time series instances, and other metadata in system tables.

CalendarPatterns

This system table stores the information about the calendar patterns, as shown in Table 1-3. The CalendarPattern data type cp_pattern defined inside the CalendarPattern system table is an opaque data type that defines the interval duration and the pattern of valid and invalid intervals in a calendar pattern.

Table 1-3 CalendarPattern table

Column name	Description
cp_name	Name of the pattern
cp_pattern	User-defined calendar pattern

In Example 1-3, the information within the curly brackets is the pattern specification. The pattern specification has one or more elements that consist of *n*, the number of interval units, and either *on* or *off*, to signify valid or invalid intervals. Elements are separated by commas.

Example 1-3 Creating calendar patterns

```
INSERT INTO CalendarPatterns
VALUES ('workweek','{5 on, 2 off} day');
```

CalendarTable

This system table stores the information about the calendars used by the database. Although most of the columns in this table are for internal use, the columns shown in Table 1-4 can be useful for you.

Table 1-4 System CalendarTable

Column name	Description
c_name	The name of the calendar
c_calendar	The Calendar type for the calendar

Example 1-4 shows how the calendar table is defined.

Example 1-4 Creating calendar

```
INSERT INTO CalendarTable (c_name,c_calendar)
   VALUES ('workweek',
```

```
'startdate(2012-01-01 00:00:00.00000),
pattstart(2012-01-01 00:00:00.00000),
pattname(workweek)');
```

TSContainerTable

TSContainer table is a system table that stores the definition of all the containers of the time series being used in the database. This table is managed by the database server, and users do not modify it directly, nor should they normally be required to view it. Rows in this table are automatically inserted or deleted when containers are created or destroyed. Table 1-5 describes the columns.

Table 1-5 System TSContainerTable

Column name	Description
name	The name of the container of the time series.
subtype	The name of the time series subtype.
partitionDesc	The description of the partition that is the container.
flags	Flags to indicate: 1. Whether the container is empty and always was empty. 2. Whether the time series is regular or irregular.
pool	The name of the container pool to which the container belongs. NULL indicates that the container does not belong to a container pool. The default container pool is named autopool.

TSInstanceTable

The TSInstanceTable contains one row for each large time series that is stored “out of row.” Time series smaller than the threshold are stored directly in a column and do not appear in this table. Table 1-6 describes the columns.

Table 1-6 System TSInstanceTable

Column name	Description
id	The serial number of the time series. This is the primary key for the table.
cal_id	The identification of the CalendarTable row for the time series.
flags	Stores various flags for the time series, including one that indicates whether the TimeSeries is regular or irregular.
vers	The version of the time series.

Column name	Description
container_name	The name of the container of the time series. This is a reference to the primary key of the TSContainerTable table.
ref_count	Number of different references to the same TimeSeries instance.

1.5.6 SQL routines and APIs

Informix provides over 100 TimeSeries-specific routines to perform SQL and other operations. It also provides a set of C APIs and Java APIs that enable you to take advantage of server library functions and develop your own routines.

The TimeSeries SQL routines create instances of a particular TimeSeries type as well as add data to or change data in it. SQL routines are also provided to examine, analyze, manipulate, and aggregate the data within a time series, which saves application development time.

1.5.7 Virtual table interface

The TimeSeries data type is a complex data type that is not easily accessible by third-party applications. Virtual tables provide interoperability with other databases and applications by displaying time series data in a relational format.

The virtual table interface (VTI) provides a relational view of the TimeSeries data and allows you to perform regular SQL operations on it. Virtual tables are useful for viewing TimeSeries data in a simple format. The virtual table is not a real table stored in the database, but just a view. The data is not duplicated. At any given moment, data visible in the virtual table is the same as the contents in the base table.

Table 1-7 shows sample data for a TimeSeries table. Table 1-8 on page 14 shows a VTI representation of the same data.

Table 1-7 TimeSeries table

Meter_id	TimeSeries column
1	[(4,160,40), (4.5,155,35), (5,165,33)….]
2	[(5,175,44), (4.5,160,35). .]
. .	. .

Table 1-8 VTI representation of the TimeSeries table

Meter_id	Timestamp	Current	Voltage	Resistance
1	2010-12-01 01:00:00.0000	4.0	160	40
1	2010-12-01 01:00:10.0000	4.5	155	35
1	2010-12-01 01:00:20.0000	5.0	165	33
. .	. .	. .	. .	. .
2	2010-12-01 01:00:00.0000	5.0	175	44
2	2010-12-01 01:00:10.0000	4.5	160	35
.	.	.	.	.

The most efficient way to work with the TimeSeries data is to carry out the data manipulation operations (INSERT/UPDATE/DELETE) directly on the base TimeSeries table using the TimeSeries API routines.

You can use the VTI to carry out some of the data manipulation operations, but the following restrictions apply:

- You cannot use UPDATE or DELETE statements on TimeSeries VTI.
- You can use SELECT and INSERT statements; however, an INSERT on the VTI table translates to an UPDATE on the underlying TimeSeries base table.
- You can update a TimeSeries element in the base table by inserting a new element for the same time point into the VTI.

Now that we have a fair knowledge about the concepts of TimeSeries and its efficient representation in Informix, we can dive deeper into its working using a user case. In the next chapter we demonstrate how to create a custom database application for a municipal electric utility company that uses the Informix TimeSeries capabilities. We show step-by-step implementation of this solution, and we also cover the database design, querying, and reporting requirements.

2

A use case for the Informix TimeSeries feature

This chapter describes the situation at a fictional company where the Informix TimeSeries feature can be used to design a database to manage complex data requirements. This use case is the basis for the examples in the remaining chapters.

© Copyright IBM Corp. 2012. All rights reserved.

2.1 Customer overview

To best demonstrate Informix TimeSeries capabilities, this example designs a database for a municipal electric utility company. The company, *HiPerfUtilCo*, serves 1 million addresses. Of the company's customers, 95% are residential customers and the remaining 5% are a mix of small and medium commercial businesses.

HiPerfUtilCo selected new advanced technology meters, enabling the company to gather meter readings every 15 minutes (that is, at *xx*:00, *xx*:15, *xx*:30, and *xx*:45). To best use the limited network bandwidth, data is to be obtained from each meter only once every 4 hours.

Residential meters gather usage and demand readings; these readings are measured in KWh and KW, respectively. Commercial meters gather usage and demand reading also, but commercial usage is measured in KVar, and demand is still measured in KW.

Usage will be gathered as both an overall total (like an odometer) and as a 15-minute total. Demand is only gathered as a 15-minute total. The advanced meters are programmable, so future reading types should be allowed for in the design if possible.

Residential meters do not apply a multiplier to the readings gathered; that is, the multiplier can be assumed to be 1. However, many commercial meters do require a multiplier to be applied to a reading to get the actual billable usage amount. Multipliers can change, but do so infrequently. Fortunately, our intelligent meters apply the multiplier as the reading is gathered, so we need to treat it as a meter attribute only.

Usage readings will be used for billing purposes. Demand readings will be used to find instances where a customer exceeds the rated capacity of their meter. When this occurs, HiPerfUtilCo will want to work with the customer to upgrade their meter or electrical service. Both numbers will also be used for Business Intelligence purposes; HiPerfUtilCo wants to try to create new billing programs and to identify the customers to target for these new or existing programs. For example, customers who use the bulk of their power during the evening might be targeted for the "Time of Use" billing program.

Billing is to be done every Monday through Friday; there are 20 billing cycles per month, and the addresses are spread fairly evenly over these 20 cycles. Each location must select a billing program. The most common billing programs are the flat rate program and the 3-tier Time of Use program. Usage is billed at the same rate all day in the flat rate program. In the 3-tier Time of Use program, usage is billed at three different rates, depending on the window of use (8 a.m. to

3:59:59 p.m., 4 p.m. to 11:59:59 p.m., Midnight to 8 a.m.). A customer can change their billing program at any time; changes are effective at Midnight on the date requested per regulatory requirements.

Billing programs must be defined to cover a complete 24-hour period, starting at Midnight local time and ending at 11:59:59 p.m. local time. Therefore, if a segment covers this time, it must be broken into two separate segments. For example, if we wanted to create a program with a 6 a.m. to 5:59:59 p.m. window, and a 6 p.m. to 5:59:59 a.m. window, we would actually need to create three segments in our database – Midnight to 5:59:59 a.m., 6 a.m. to 5:59:59 p.m. and 6 p.m. to 11:59:59 p.m..

HiPerfUtilCo is also interested in expanding its use of Geographic Information Systems (GIS) and GPS data; therefore, it would like each meter's actual GPS location to be stored. This is *not* the location address GPS but the GPS where the meter is actually installed on the property. This could be useful, for example, to find all meters within a certain distance of lightning damage, so that they can be inspected or queried. This can also be used to find all customers within a specific area for advanced analytical purposes such as load planning or bill analysis.

One major issue that HiPerfUtilCo has had to deal with over the years is daylight saving time. The IT systems within HiPerfUtilCo are typically GMT-based for this reason. From a metering standpoint, it means that 96 readings are captured on most days; however, on one day per year, only 92 readings will be captured; and one day per year 100 readings will be captured. The meter is aware of the current time because the Public Utility Commission requires that meters display the current date and time when queried. The 4-hour block of readings is time-stamped only with the start of the block and the readings within are then just sequenced, so the Midnight block covering 2 a.m. will have 12, 16, or 20 readings sequenced within it as required.

Normally, utilities use a validate, edit, and estimate (VEE) process against the raw meter data to validate or correct the data for billing purposes. This process is a complex subject that is beyond the scope of this document. However, some aspects of the VEE process can be addressed. For example, one key aspect of VEE is to locate missing reads and to create estimates for these gaps based on historical data for the customer or the location or from comparable accounts. For details, refer to 5.3.1, “Creating a custom function” on page 61.

2.2 The initial data model

Based on the customer requirements, the scenario for this book must include a hypothetical OLTP data model for the utility. You can design a schema without a

time series data type and then use common patterns to modify it for a time series data type.

For meters, the data model uses the serial number and manufacturer as the natural key and stores the meter's status and multiplier as additional attributes.

For locations, create a surrogate key because there is no natural key. The data model needs the type of location (for example, residential or commercial as *R/C*). The data model can store the address fields as attributes. In addition, it can store the GPS location of the meter at this location.

For customer, add a surrogate key (for example, the account number) and store attributes, such as name fields and phone number fields, and a tax identifier, such as a social security number (SSN).

For the billing program, the data model only needs a surrogate key and the program name. Create a second table to hold the program details. This table uses the program's surrogate key plus a segment number as the keys. Each segment has a particular start and end time of day and the rate to be charged during this time.

For meter readings, create a separate table that can store all of the readings that are collected for each location and time stamp. Effectively, the data model associates the readings only to the location, regardless of the customer or meter, to facilitate analysis by location.

Finally, create relational tables between these tables to understand which meter is installed at which location, including when it was installed and the starting read when installed. You need to know which customer is associated with each location, along with the type of relationship (for example resident, landlord, and other information). Finally, you need to know which billing program is in effect at each location and when it began.

Figure 2-1 on page 19 illustrates the starting schema.

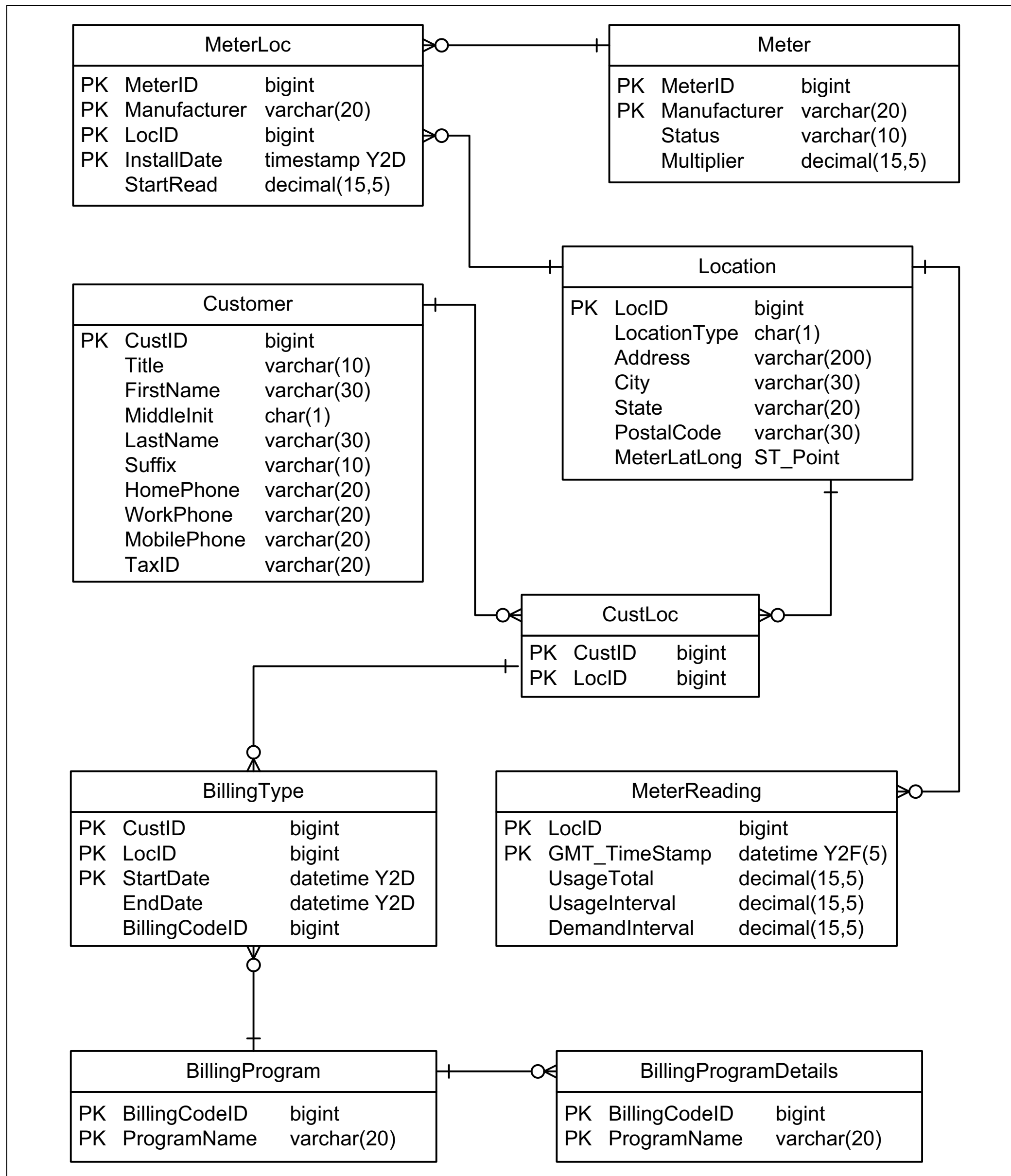


Figure 2-1 Initial schema

Note that this scenario provides a simple model and does not necessarily show the complexities of the relationships faced by a real-life utility. Relationships might face the following real-life complexities, among others:

- Many time-of-use billing programs also have a day-of-week component (for example, Monday through Friday is one rate and weekends are a different rate) or a time of year component (for example, from June to September has a different rate during the day than does the remainder of the year).
- The model somewhat allows for one customer to be associated with multiple locations, which can cover situations where, for example, a customer has a home and a business and wants the business bill sent to the home address. However, the model does not allow for multiple customers to be associated with a single location, which can happen, for example, where a landlord is supposed to receive a bill when a tenant fails to pay the bill.
- The model also simplifies the relationship between a location and a meter. It expects that each location has only one meter, although it does allow for that meter to be replaced if needed. Some real-world locations can have multiple meters. Most locations, such as apartment buildings, can simply be treated as separate locations for billing purposes, and unique locations are, therefore, created in the table. However, some complex situations exist in the real world that this model simply cannot represent.

  Here is one example: In rural areas of Texas, the meter for a farm or ranch is initially installed nearer to a pole near the road than to the house, making it easier to read. Sometime later, an oil well is installed midway between that pole and the home with a separate meter, which someone else pays. However, the first meter now shows the usage for the home and this oil well. To correctly compute the bill for the house, the second meter’s reading must be subtracted from the first. Many utilities are working to eliminate these situations.

2.3 Applying a time series data type to the relational data model

To apply a time series data type to the relational data model, begin by identifying the different time stamps in the model to determine the best candidates for use with a time series data type:

- The MeterLoc table has a time stamp, but meters do not move around that time stamp often. Thus, the number of dates produced by this time stamp is actually small and this table is not a good candidate for use with a time series data type.
- The BillingProgramDetails table has two time stamps, but these time stamps represent times of day only (for example, 8:00 a.m. or 12:00 p.m.). Thus, this table is not a good candidate for use with a time series data type.
- The MeterReading table is going to end up being large, with an average 96 entries per day per location reporting. Because these entries are also at regular time intervals (every 15 minutes), a regular time series data type works nicely with this table.
- The BillingType table is another table with time stamps. Although most customers do not change billing programs often, the pattern of using a starting and ending time stamp indicates that an irregular time series data type can be used effectively. The end time of one program selection corresponds to the start time of the next one.

After you know the candidate time series data type, you can then pull out the data for the time series data type into a *Row Type*. The Row Type must start with a time stamp. For the MeterReading table, add the three reading values that are captured in each interval. For the BillingType table, you only need the Billing Program ID.

Figure 2-2 on page 22 shows the modified schema.

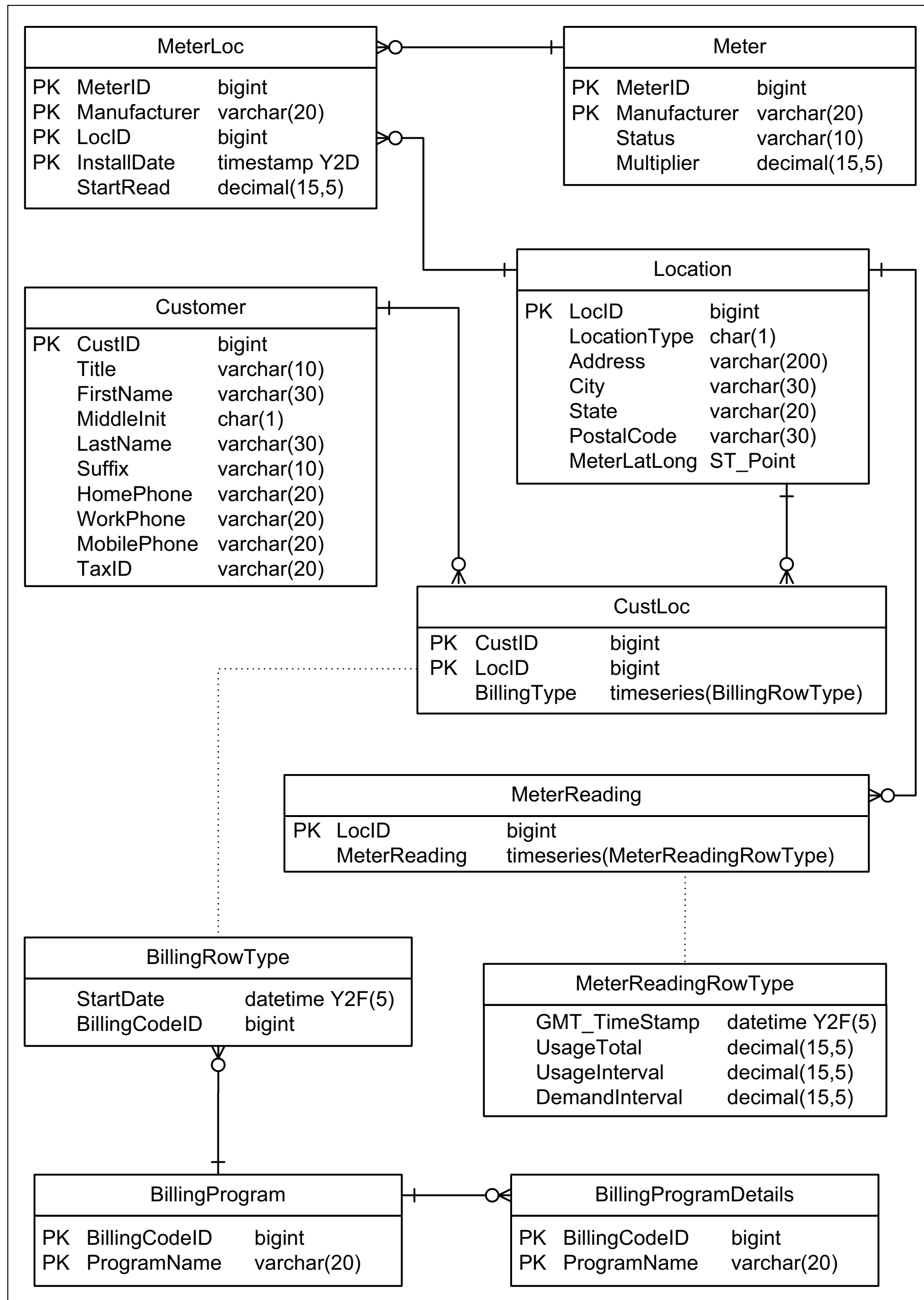


Figure 2-2 Time series data type schema

2.4 Final considerations

Before finalizing the schema, consider the following additional factors:

- A row type cannot be changed after it is in use.

  If you think that more reading types will be needed in the future, consider adding those fields now and leaving them null until they are needed.

  The alternative is to create new row types in the future to hold different reading values. If you plan to do this, you need a way to know which row type is in use in any particular time series data type. So, consider adding a row type indicator to the table now.

- There might be performance implications when using the Informix TimeSeries feature.

  A time series data type can store the row data into the table's row (for example, in CustLoc) or into a separate container, which requires additional I/O operations to access. As a general guideline, in-row storage should not exceed 1500 bytes.

  The BillingType table time series data type is expected to have few entries. So, store it in-row if possible. Specify a threshold value of 65 to maximize the number stored in-row. However, if a time series data type exceeds this value, it moves to a container, causing additional I/O operations.

  For the MeterReading table, however, simply specify a threshold of zero (0), which forces all entries to a container. This value does mean at least 1 I/O is needed to access any MeterReading table time series data.

3

Defining your TimeSeries environment

There are many things to consider when defining your TimeSeries environment. This chapter reviews the different technologies and common issues related to time series data processing.

© Copyright IBM Corp. 2012. All rights reserved.

3.1 Schema

Time series data does not exist in a vacuum. It exists with a lot of other information that is likely to be stored in relational format. Thus, you have to design a schema that is appropriate to optimize your business processes on this data. Even without considering the time series data, you can do a standard relational schema.

Figure 3-1 illustrates a high-level table relationship.

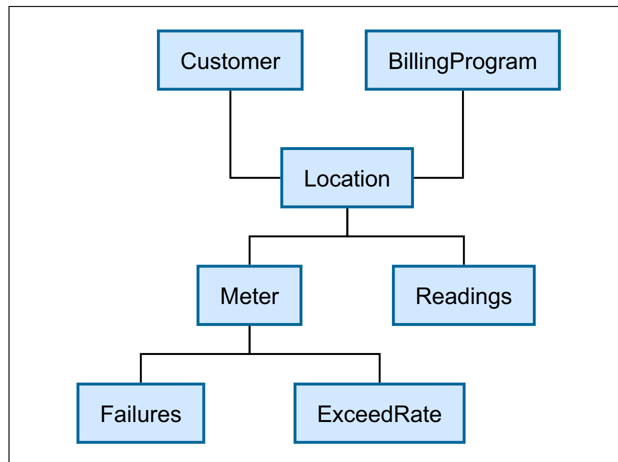


Figure 3-1 Simple table relationships

This figure shows that a customer has a location, which might be multiple locations. A location has a billing program, a meter, and a set of readings. A meter then has failure events attached to it and events that indicate when the meter has exceeded its rate capacity.

This illustration is actually a simplification of a schema that was used, and it clearly demonstrates this schema's issues. For one, the events that are attached to the meters should really be attached to the location. For this scenario, it is more important to see the overall picture at the location level than at the meter level. A meter can be replaced, but regardless of which meter is there, the events matter.

The other thing to consider is the way that the system will be used. The Readings table is by far the largest table in the system. It is also the table that will be at the center of most of the major queries. You want to make sure that you get to the readings while involving as few table joins as possible.

Another important point to consider is the multiplier that was mentioned as a meter attribute in the previous chapter. This multiplier can be used in two ways:

- To multiply the Readings values when operating on the meter readings
- To do the multiplication at insert time and then use it to divide if you ever want to retrieve the exact meter readings

The second approach is preferred because then the multiplier is not involved in any analysis and billing queries, which reduces the processing load and simplifies the queries.

There are further consideration, which we discuss later in this chapter. However, at this point, our schema is changed as shown in Figure 3-2.

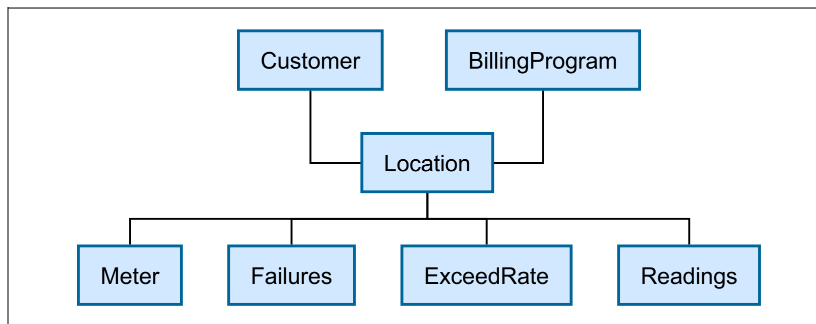


Figure 3-2 Modified table relationships

Some people jump to the conclusion that because there could be a large number of transactions on the systems, it should be following an ER diagram and go through the normal forms to optimize the database. It might be the case that normalizing is an efficient way to represent the data, but do not jump to this conclusion automatically. Consider other types of models, such as the star (snowflake) schema or even a hybrid that has portions as a star schema and other portions that are more of an ER model. It all goes back to modeling and considering the different use cases that constitute the core of the system.

A majority of the queries will be concerned with manipulating the readings (meter data). As mentioned earlier, you want to get to the Readings table with a minimum of I/O operations. You might want to locate a set of meters based on their geographical location, such as proximity to a specific location, or through political boundaries, such as city and county. Looking at the queries that you require will possibly lead to some sort of hybrid model. Whether you need a hybrid model will be more evident after you see how time series data fits in the solution.

You are the ultimate judge on how to organize your data. The good news is that all you know about database design still applies here.

3.2 Relational and time series data working together

The Informix TimeSeries feature includes a data type that is part of the relational framework, which means that this data type is simply another column type available to use in table definitions. For example, you can select the meter data on which you want to operate based on standard relational attributes (columns), including spatial information, as highlighted in the following query:

```
SELECT GetFirstElem(TSRollup( AggregateBy('sum($value)', 'ts_1day',
raw_reads, 0, '2010-11-10 00:00:00'::datetime year to second,
'2010-11-10 23:45:00'::datetime year to second),
'avg($value)')::timeseries(meter_data), 0).value
FROM ts_data t1, ts_data_location t2
WHERE t1.loc_esi_id = t2.loc_esi_id
AND ST_Contains('4 POLYGON ((-122.28343657141261 37.43915215530022,
-122.28343657141261 37.54157880984285, -122.08293608313136
37.54157880984285, -122.08293608313136 37.43915215530022,
-122.28343657141261 37.43915215530022))', longlat)
```

The `timeseries` of this query comes from the `ts_data` table (from the stores demo database that comes with Informix) and has the following definition:

```
CREATE TABLE ts_data (
   loc_esi_id       char(20) NOT NULL,
   measure_unit     varchar(10) NOT NULL,
   direction        char(1) NOT NULL,
   multiplier       TimeSeries(meter_data),
   raw_reads        TimeSeries(meter_data),
   PRIMARY KEY(loc_esi_id, measure_unit, direction)
) LOCK MODE ROW;
```

This definition shows that the `raw_reads` column is of type `TimeSeries` and subtype `meter_data`.

The TimeSeries subtype, `meter_data`, defines the elements in each reading of the time series columns. It has the following definition:

```
CREATE ROW TYPE meter_data (
   tstamp     datetime year to fraction(5),
   value      decimal(14, 3)
);
```

This query calculates a day average of a set of meters selected based on a longitude and latitude component. The query joins the customer location information table with the table that includes the time series data type. It takes advantage of the spatial capabilities of Informix to locate the customers that are included in the polygon provided. When selected, it aggregates the day values for 15-minute intervals to a day value. All the day values are then averaged together using the `TSRollup` aggregation function. If needed, you could actually rewrite this query to use the `AVG` relational aggregate function.

The main thing to remember is that standard SQL queries identify the subset of rows containing the time series data type on which you want to operate. Thus, all you know about relational databases is relevant in this environment.

For example, although meter data is kept in a time series data type, what about the remaining data? What type of data belongs in a time series data type and what type of data belongs in standard relational format? In general, if data is not time-based, it is relational. It becomes tricky when data is time-based but represents occasional values.

If you expect just a few occasional values, it might be fine to put the data in relational format. For example, if power failures are rare, you might want to keep data regarding power failures in relational format because the table will not be large. If, however, the power grid is not reliable and you want to keep power failure information for a number of years, it might be worth it to keep the data in a time series data type.

The threshold of when to use regular relational tables and when to use time series data types is difficult to pinpoint. Much of the decision has to do with how the data will be used. Time series data type operations can make it easier to get to the answer you want.

3.3 Regular or irregular data

As described in Chapter 1, “Overview of IBM Informix” on page 1, the Informix TimeSeries features supports both *regular* and *irregular* interval data. Because the smart meter readings come at regular intervals, you can use regular time series data types for storage. Referring back to Figure 3-2 on page 27, the modified table relationships have the following representation for table readings:

```
CREATE TABLE readings (
    LocId           bigint,
    MeterReading    TimeSeries(row_meterReading)
);
```

This is the same definition as provided in Chapter 2, “A use case for the Informix TimeSeries feature” on page 15. In this use case, the Location and Readings tables join together using the `LocId` primary keys. All the readings for a specific location are found in one row.

Also shown in Figure 3-2 on page 27 is the potential for multiple time series data types, including `Failure`, `ExceedRate`, and `BillingProgram`. `Failure` and `ExceedRate` can include a begin and an end of event, which is perfect for irregular time series data types because, usually, there are no events.

The approach for `BillingProgram` might not be that clear. In chapter 2, we mentioned a 3-tier program with which you can use regular time series data types with a calendar that divides the day into three parts.

This approach can also be accomplished using an irregular time series data type. Consider the billing program that uses a flat rate. Using a time series data type representation, either regular of irregular, can make it easier to manipulate the data, but it might make more sense to keep the data in relational format and use a stored procedure or a custom function when doing the billing. Chapter 5, “Querying TimeSeries data” on page 53 shows an example of this type of approach.

The effort is complicated by the fact that the billing program can be changed on a daily basis. This type of change represents an extreme case, but at a minimum the model needs to accommodate the possibility of having multiple billing programs that apply to one billing cycle.

The model also needs to consider corner cases, where a customer starts in the middle of a billing cycle or ends before the end of a billing cycle. This circumstance should be fine because the Readings table will not have data for those days.

One further consideration related to `BillingProgram` is how to keep track of the billing program changes over time for each location. The answer comes from how efficient you can make the billing processing. There are two possibilities currently, but there might be additional billing programs in the future. What if the billing programs change over time? You have to keep track of changes for historical purposes. It might be easier to create a new billing program than to try to keep track of the changes over time.

After these questions are answered, you still need to be able to do the billing. The most difficult part is actually determining which rate applies when. In particular, how do you figure out the rate that applies at the beginning of the billing period? This rate might have been picked months earlier. To solve that problem, you can store the billing rate in an irregular time series data type, as is done in the case

described in chapter 2. Then you have to go through each billing day and apply the appropriate billing schedule.

3.4 The TimeSeries element

A TimeSeries element involves multiple objects. Figure 3-3 summarizes the relationships involved.

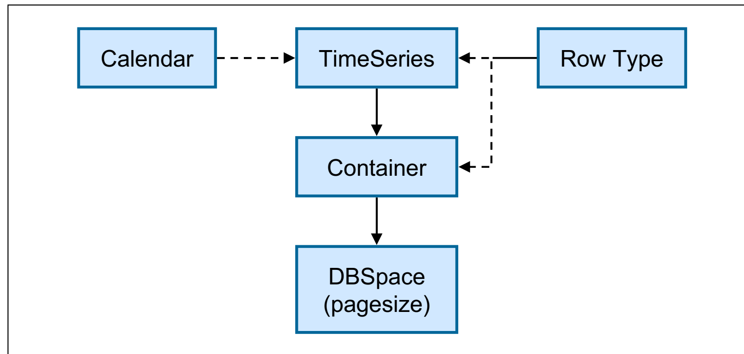


Figure 3-3 TimeSeries objects relationships

A TimeSeries element is built from a row type definition. The Informix documentation refers to it as the *subtype* of the TimeSeries data type. A TimeSeries element is stored in a container that uses the same subtype as the TimeSeries data type. There is an additional restriction that regular and irregular TimeSeries elements cannot be created in the same container.

The container uses space from database storage called a *dbspace*. A dbspace is created using a specific page size that can be a multiple of the default page size (2 KB or 4 KB) up to 16 KB.

A TimeSeries element is limited to around 32 KB, the same limit as the maximum size of a row in a table. There are a few data types that are not supported, such as bigserial, serial, and opaque types that have an out-of-row representation. There is also a limit of 255 elements per page. For a regular TimeSeries element, the time stamp is not stored with the element. Thus, for an element that contains only one float (8-byte) value, make sure that you use a container that has a 2 KB page size. In the case study described in chapter 2, the element size is around 42 bytes. It could use up to an 8 KB page size without wasting space.

Look carefully at what you want to put in a TimeSeries element. Some people consider putting the billing rate in the element. Looking back at the earlier discussion on billing programs, you can see that it makes it much easier to do billing using that approach. The drawback is that the same value is repeated over and over. The repetition of this double precision value (8 bytes) adds up to a lot of space. For example, consider that if you have one million meters collecting data at 15-minute intervals for one year, that additional field in the TimeSeries element ends up using around 280 GB in disk space.

The first thing to consider when defining the content of a TimeSeries element is whether you really need this value in each element. Equate this decision to the standard schema normalization for relational databases. Then, just like in the case of normalization, look at how the data will be used and whether you want to trade in space for faster processing.

It is possible to include strings in a TimeSeries element, but keep in mind that, no matter how small it is, it includes a 4-byte value to keep track of its length. Thus, a 1-byte flag will use 5 bytes. Consider using a numerical representation of that flag and storing it in a `SMALLINT` that uses two bytes.

In general, use mostly numeric types in a TimeSeries element, and perhaps date and datetime types. Still, you have to decide which types to use, such as `SMALLINT`, `INT`, `BIGINT`, `DECIMAL`, `REAL` (`SMALLFLOAT`), or `DOUBLE PRECISION` (`FLOAT`).

Some of these types are called *exact numeric* (`SMALLINT`, `INT`, `BIGINT`, and `DECIMAL`), and others are *approximate* (`REAL`, and `FLOAT`). Choose the type carefully because of the operations that you will do on them. The data type returned in any operation is the same data type coming in by default.

Ask yourself the following questions:

- Is the type big enough?

  If you are summing up a bunch of values, make sure the result can fit in the data type selected.

- Is the type precise enough?

  If you use a decimal type with a precision of 2 digits after the decimal point, is it precise enough for an answer back after operating on a number of values? For example, if you are doing an average.

The use case described in chapter 2 uses decimal types for the usage and demand values in the TimeSeries element. It provides a high level of precision and flexibility and makes for a great choice in this case.

The Readings TimeSeries element is defined as follows:

```
CREATE ROW TYPE MeterReadingRowType(
   GMT_TimeStamp        datetime year to fraction(5),
   UsageTotal           decimal(15,5),
   UsageInterval        decimal(15,5),
   DemandInterval       decimal(15,5)
);
```

In the course of defining the operations that you will perform on your time series data, you might have to define new element definitions for the result of some time series operations. For example, if you are aggregating multiple time series data to get a usage total, you do not need to return the three numerical values shown in the previous example but, instead, create a new TimeSeries element type that contains only the values that you want in the resulting element. Chapter 5, “Querying TimeSeries data” on page 53 shows further examples.

3.5 Calendars

Calendars are important because they indicate when an interval is valid for regular time series data. Because readings are expected in all valid intervals, calendars are useful to indicate which intervals are stored. Calendars are also used for irregular time series data, but are usually kept simple because there is no real concept of valid intervals in that case.

Informix comes with a set of useful calendars that includes 1 minute, 15 minutes, 30 minutes, 1 hour, 1 day, 1 week, and 1 month intervals. These intervals might be all that you need.

Weekly calendar: The weekly calendar that is defined with Informix begins on a Monday. You might prefer to have a weekly calendar that begins on a different day, such as a Sunday, or you might want a weekly calendar that is active only during the weekdays.

For example, if you want a weekly calendar where only the weekdays are active, you need to use a pattern where you indicate the weekdays being on and weekends being off. If you start on a Monday, the pattern would be 5 on and 2 off, as follows:

```
INSERT INTO CalendarTable(c_name, c_calendar)
VALUES (’weekday’, 'pattstart(2012-01-02 00:00:00.00000), pattern({5
on, 2 off} day)');
```

3.6 Time zone issues

If you need to collect data in different time zones, normalize the data to Coordinated Universal Time (UTC). The Informix TimeSeries feature does not support time zones. Many time zones use daylight savings time, which has two irregular days in the year: one day with 23 hours and another day with 25 hours. Using UTC keeps all days at 24 hours. You can decide in your application how to deal with the time changes during the year.

3.7 Loading data

One of the key operations in any system is the loading of data. Informix TimeSeries systems usually manage a large amount of data. You also need to consider the timing requirements of updates. You might need to load the data as it arrives, or you might collect the data in 1-hour, 4-hour, or daily increments before loading it. Currently, the faster means of loading data favors collecting data for an amount of time before loading it.

At the time of this writing, the following loading methods are available:

- Insert data through the relational view of the Informix TimeSeries virtual table interface (VTI)
- Update TimeSeries data using the `PutElem()` function
- Update TimeSeries data using the `BulkLoad()` function to load the content of a file
- Use the Informix TimeSeries plug-in for Data Studio

More options might become available in the future. Make sure to review the release notice and the documentation for the latest release of Informix for the most up-to-date information. You can also find information at the IBM developerWorks Informix smart meter central wiki at:

https://www.ibm.com/developerworks/mydeveloperworks/wikis/home?lang=en#/wiki/Informix%20smart%20meter%20central/page/Creating%20and%20loading

The faster loading method is using the Informix TimeSeries plug-in for Data Studio. It is described in the TimeSeries documentation. Data Studio is a product that is available on the Informix media distribution and for download at no charge from the IBM website. The plug-in allows you to define the format of the input file and map it to a specific table. An input file record includes a unique identification of the target TimeSeries (primary key) and the values for the TimeSeries elements, including the time stamp. With a definition of an input file format, TimeSeries table, and the mapping between the two, you can load your data efficiently.

3.8 Data cleansing and corrections

Data cleansing is also referred to as *validation, estimation, and editing* (VEE). It involves operations such as checking for missing values and out of range values. In the E&U industry, the rate of error is in the range of one to two percent.

Depending on the volume of data being loaded and the size of your system, you might want to do part of your cleansing in the data preparation step. You might have quite a few systems that collect the data and then feed the data to the database system for injection. Doing some of the cleansing up front can reduce the demand on the database system.

Your loading processes can also keep track of the meters that might now include suspicious data. Then you can use a second step to complete the cleansing process. If you already corrected the simpler errors, you can cover much less meter data in the second step.

3.9 High availability

Informix has a feature called *high availability data replication* (HDR). This feature allows you to set up one or more secondary servers that can take over in the case of a failure of the primary system. The data is replicated automatically to that secondary server. The servers do not need to be co-located. They can even be on different continents if your network backbone is fast and solid. The secondary server can also be active at the same time as the primary server, which is a way to distribute the processing, such as billing, over multiple system if necessary.

For more information, see the *IBM Informix Administrator's Guide*, starting at Chapter 21. You can find information about where to find this manual in Appendix A, "Reference material" on page 101.

3.10 Backup

Informix provides the capability to do both full and incremental backups that can be augmented with the use of other storage management software. You want to back up the non time series tables because customer and location information can change. It might also make sense to back up the corrections made to the data. You might decide to back up the raw data files instead, and have a plan to reload them if needed. Consider all of these possibilities as you define your backup strategy.

3.11 Purging data

In most cases, your data will grow by adding newer data to your TimeSeries environment. You will reach a point where you want to remove the older data from your system. The TimeSeries API currently provides the following functions to delete a range of data:

- `DelClip`
- `DelRange`
- `DelTrim`

In most cases, the `DelRange` is sufficient. For more information, see the *IBM Informix TimeSeries Data User's Guide*.

Keep in mind that TimeSeries functions do not allow intra-query parallelism. If you are planning to remove the older data from the whole TimeSeries environment, you will have a considerable number of rows containing TimeSeries data type columns on which to operate. Consider starting multiple client processes or threads, each with its own database connection, and have each connection operate on a subset of the data.

The biggest consideration for purging data is the logging capacity of your system. Make sure that you have enough logging space to handle the operations. One way to limit the logging space is to increase the frequency of checkpoints. Also consider how fast the logging subsystem is. For example, if your logging is done on solid-state drives (SSD), the system can handle a larger number of purging processes.

3.12 Spatial requirements

Spatial information is included in quite a bit of data. For example, an address is spatial information but might not provide enough information. It is better to augment an address with a longitude and latitude.

Informix supports spatial queries as a standard function that comes with the Informix server. You can add spatial objects, such as point, line, and polygon, to relational tables. For example, you can use a polygon to define the limits of a county and retrieve the readings that are at a location within that county. This function opens the door to many possibilities for grouping and reporting.

4

Implementing Informix TimeSeries

This chapter transforms the schema diagram from Figure 2-2 on page 22 to a working Informix schema. The examples included in this chapter compute space requirements for both relational and time series storage, compute the possible space savings, and discuss techniques for loading the time series data.

© Copyright IBM Corp. 2012. All rights reserved.

4.1 Schema definitions

The time series schema diagram from Figure 2-2 on page 22 must be converted to SQL definitions. Although the majority of the conversion is straightforward, keep in mind the following considerations:

- The row types must be defined before using them to define the time series columns.
- The time series columns must be defined before the virtual table interface (VTI) routines are called.

4.2 Space calculations

For space calculations, first compute how much space is required by the relational tables and relational portions of tables holding time series columns. Then, add the space that is required by any indexes. Finally, add the space required by the time series columns.

> **Temporary space:** These calculations do not include temporary space for queries, nor do they account for growth of the data over time (for example, the number of customers increasing each year).

4.2.1 Space calculations for relational storage

The space calculation for the relational tables and relational portions of tables holding time series columns is well documented in the *IBM Informix Performance Guide, Version 11.7,* SC27-3544.

This section includes tables that list the space calculation values that are used for the relational tables. Each table is followed by definitions of the calculated columns in that table. These calculations show that about 7.2 GB of storage are needed for the data.

Table 4-1 gathers row size and row counts for each relational table as follows:

- The average size for the location table was computed by allowing the VARCHAR fields to be 60% of their maximum sizes.
- Small row sizes under 100 bytes were not adjusted for average size, even though they might contain VARCHAR fields.

Table 4-1 Table size information

Table name	Max row size	Avg row size	Number of rows
billingprogram	29	29	100
billingprogramdetails	29	29	300
meter	49	49	1,000,000
meterloc	51	51	1,000,000
customer	177	107	1,000,000
meterreading	2,060	2,060	1,000,000
custloc	2,068	2,068	1,000,000
location	2,344	2,232	1,000,000

Table 4-2 shows the following page size calculations. The assumption is that a 2 KB page size is used for all tables. 2 KB is the default page size for most systems, but 4 KB is the default for IBM AIX® and Windows operating systems.

- Using sizes higher than 2 KB pages for billingprogram, billingprogramdetails, meter, meterloc, and customer doubles the space required.
- Using sizes higher than 2 KB pages for billingprogram, billingprogramdetails, meter, meterloc, and customer on systems with a default 2 KB page size also doubles I/O needed to read in the data.
- Using 4 KB pages for the meterreading, custloc, and location tables nearly doubles the space required but could be I/O neutral over a 2 KB page. 4 KB of data is required in both cases to read a single row.
- Using 8 KB pages for the meterreading, custloc, and location tables increases the space required by approximately 13% but doubles the I/O to read a single row over 2 KB or 4 KB pages.
- Using 16 KB pages for the meterreading, custloc, and location tables decreases the space required by approximately 3%, but quadruples the I/O to read a single row over 2 KB or 4 KB pages. The space savings are minimal against the I/O costs.

Table 4-3 shows the calculations for tables whose row size is less than the page size available. Table 4-4 and Table 4-5 show the calculations for tables whose row size is greater than the page size available. The calculations reference the following functions:

- ROUNDUP rounds the value to the nearest integer higher.
- ROUNDDOWN rounds the value the nearest integer lower.
- MIN returns the lower of its arguments.

Table 4-2 Page size calculations

Page size	Page use[a]
2048	2020

a. Page Use = Page Size - 28

Table 4-3 Tables with row size less than Page Use size

Table name	Rows per page[a]	Total pages needed[b]	Total space needed (bytes)[c]	Total space needed (MB)[d]
billingprogram	61	2	4,096	< 0.01
billingprogramdetails	61	5	10,240	0.01
meter	38	26,316	53,895,168	51.4
meterloc	36	27,778	56,889,344	54.3
customer	18	55,556	113,778,688	108.5
TOTAL				*214.2*

a. Rows per page = MIN (255, ROUNDUP (Page use/(Avg row size + 4)))
b. Total pages = ROUNDUP (Rows per page/Number of rows)
c. Total space needed (bytes) = Total pages * Page size
d. Total space needed (MB) = Total space needed (bytes)/(1024 * 1024)

Table 4-4 Tables with row sizes greater than page use

Table name	Home pages[a]	Remainder size[b]	Partial remainder size[c]	Partial ratio[d]
meterreading	1,000,000	32	48	0.02
custloc	1,000,000	40	56	0.03
location	1,000,000	204	220	0.11

a. Home pages = Number of rows
b. Remainder size = Avg row size - (Page Use + 8)
c. Partial Remainder Size = MOD(Avg row size, Remainder size - 8)
d. Partial ration = Partial remainder size / Page Use

Table 4-5 Tables with row sizes greater than page use (continued)

Table name	Partial remainder pages	Total pages needed[a]	Total space needed (bytes)[b]	Total space needed (MB)[c]
meterreading	142,858[d]	1,142,858	2,340,573,184	2232.1
custloc	166,667[d]	1,166,667	2,389,334,016	2278.6
location	250,000[e]	1,250,000	2,560,000,000	2441.4
TOTAL				**6952.1**

a. Total pages needed = Home pages + Partial remainder pages
b. Total space needed (bytes) = Total pages * Page size
c. Total space needed (MB) = Total space needed (bytes) / (1024 * 1024)
d. Partial remainder pages [meterreading and custloc] =
ROUNDUP(Number of Rows / (TRUNC((Page Use / 10) / Remainder size) + 1))
e. Partial remainder pages [location] =
ROUNDUP(Number of Rows / (TRUNC((Page Use / 3) / Remainder size) + 1))

4.2.2 Space calculations for indexes

Each table has a primary index. The space calculation for indexes is also well documented in *IBM Informix Performance Guide, Version 11.7,* SC27-3544.

This section includes tables that list the calculations for these indexes. The calculations show that an additional 212 MB is needed.

Table 4-6 shows the length of the index and starts the calculations. Table 4-7 computes the leaves and branches needed. Table 4-8 computes the space needed. To simplify the calculations, it is assumed that the tables are not fragmented, and a 90% fill factor is used.

Table 4-6 Index size information

Table name	Index length	Entry size[a]	Page entries[b]	Node entries[c]
billingprogram	12	21	96	130
billingprogramdetails	20	29	69	88
meter	29	38	53	65
meterloc	50	59	34	31
customer	12	21	96	130
meterreading	12	21	96	130
custloc	24	33	61	76
location	12	21	96	130

a. Entry size = Index length + 4
b. Page entries = ROUNDDOWN (Page use/Entry size)
c. Node entries = ROUNDDOWN (Page use/(Index length + 4) + 4)

Table 4-7 Index size calculations for leaves and branches

Table name	Leaves[a]	Branches_0[b]	Branches_1[c]	Branches_2[d]
billingprogram	2	1	-	-
billingprogramdetails	5	1	-	-
meter	18,868	291	5	1
meterloc	29,412	718	18	1
customer	10,417	81	1	-
meterreading	10,417	81	1	-
custloc	16,394	216	3	1
location	10,417	81	1	-

a. Leaves = ROUNDUP (Number of rows/Page entries)
b. Branches0 = ROUNDUP (Leaves/Node entries)
c. Branches1 = ROUNDUP (Branches0/Node entries)
d. Branches2 = ROUNDUP (Branches1/Node entries)

Table 4-8 Index size calculations for pages

Table name	Compact pages[a]	Index pages[b]	Total space needed (bytes)[c]	Total space needed (MB)[d]
billingprogram	3	4	8,192	0.01
billingprogramdetails	6	7	14,336	0.01
meter	19,165	21,295	43,612,160	41.6
meterloc	30,149	33,499	68,605,952	65.4
customer	10,499	11,666	23,891,968	22.8
meterreading	10,499	11,666	23,891,968	22.8
custloc	16,614	18,460	37,806,080	36.1
location	10,499	11,666	23,891,968	22.8
TOTAL				**211.5**

a. Compact pages = Leaves + Branches0 + Branches1 + Branches2
b. Index pages = ROUNDUP((100 * Compact pages) / 90)
c. Total space needed (bytes) = Index pages * Page size
d. Total space needed (MB) = Total space needed (bytes) / (1024 * 1024)

4.2.3 Space calculations for Informix TimeSeries storage

The space calculation for Informix TimeSeries storage is similar to that for relational data. However, keep in mind the following key points:

- For regular time series, the datetime field at the beginning of the row type is not stored, so space calculations can remove this data. For irregular time series, however, the datetime field at the beginning of the row type is stored and cannot be removed. In both time series types, all other fields are stored.
- Similar to relational tables, only 254 elements can be stored on one page.
- Unlike relational tables, no two time series instances will ever share a page from a container at any time.

One final consideration not already discussed is the number of readings to retain, which is needed to complete the space calculations of the reading data. The customer has stated that 25 months of data must be retained; that is, the current partial month plus the past 24 complete months.

This requirement results in 96 * (2 * 365.25 + 31) = 73,104 entries to be retained for the regular meter reading time series. For the billing program irregular time

series, a worst case scenario where the customer changes the program every single day for the same 25 month program results in 762 retained items.

Tables showing the calculations follow. The calculations show that 1.95 TB are needed. Table 4-9 shows the row type lengths used in the calculations. Table 4-10 computes the number of pages needed for a single time series instance. Table 4-11 computes the total number of pages needed.

Table 4-9 Time series size information

Table name	Time series type	Regular?	Defined row length	Use row length
meterreading	MeterReadingRowType	Yes[a]	37	27
custloc	BillingRowType	No[b]	19	19

a. If Regular: Use row length = Defined row length - 10
b. If Irregular: Use row length = Defined row length

Table 4-10 Time series page calculations for a single time series

Table name	Time series type	Items to retain	Maximum items per page[a]	Pages to hold 25 months of items[b]
meterreading	MeterReadingRowType	73,104	74	988
custloc	BillingRowType	762	106	8

a. Maximum items per page = MIN (ROUNDDOWN(Page Use/Use Row Length), 254)
b. Pages to hold 25 months of items = ROUNDUP(Items to retain/Maximum items per page)

Table 4-11 Time series space calculations

Table name	Number of time series[a]	Total pages needed[b]	Total space needed (bytes)[c]	Total space needed (MB)[d]
meterreading	1,000,000	988,000,000	2,023,424,000,000	1,929,688
custloc	1,000,000	8,000,000	16,384,000,000	15,625
TOTAL				**1,945,313**

a. Number of time series = Number of rows (from relational calculations)
b. Total pages needed = Number of time series * Pages to hold 25 months of items
c. Total space needed (bytes) = Total pages needed * Page size
d. Total space needed (MB) = Total space needed (bytes) / (1024 * 1024)

4.2.4 Container calculations for Informix TimeSeries storage

Similar to fragmented relational tables, Informix TimeSeries performance is aided by spreading the data across multiple containers. For 2 KB pages, the maximum container size is 32 GB (16,384,000 pages).

It is also useful to multithread queries or loading activities across containers. Ideally, you want to have at least as many containers as CPUs. However, it might not be practical to manage many small containers. Setting a lower number of containers might achieve at least some multithreading, which might be useful.

You can now compute the number of containers that are needed for time series storage. You can use trial and error to determine the best container size to achieve a given lower limit on the number of containers. Table 4-12 shows these results.

Table 4-12 Time series container calculations

Table name	Time series type	Container size (pages)	Containers needed
meterreading	MeterReadingRowType	16,384,000	60
custloc	BillingRowType	1,024,000	8

4.2.5 Time series storage needs versus relational storage needs

1.92 TB sounds like a lot of space until you contrast this amount with what it would take if you did not use time series. Assume that the first relational schema shown in Figure 2-1 on page 19 was implemented instead of time series. The calculations in the tables that follow show that the same 25 months of reading data would require nearly 3.5 TB of storage.

Table 4-13 shows the row size for relational space computations. Table 4-14 computes the space needed.

Table 4-13 Table size information

Table name	Max row size	Avg row size	Number of rows
$\text{meterreading}_{orig}$	45	45	73,104,000,000
custloc_{orig}	16	16	1,000,000
$\text{billingtype}_{orig}$	46	46	761,500,000

Table 4-14 Tables with row size less than page use size

Table name	Rows per page	Total pages needed	Total space needed (bytes)	Total space needed (MB)
$meterreading_{orig}$	41	1,783,024,391	3,651,633,952,768	3,482,470
$custloc_{orig}$	101	9,901	20,277,248	19.3
$billingtype_{orig}$	40	19,037,500	38,988,800,000	37,183.6
TOTAL				**3,519,671**

4.3 Loading time series data

Data can be loaded into time series columns using the methods described in this section.

4.3.1 Small amounts of data

For small amounts of data, you can use the Informix TimeSeries API functions to update a time series element. In the use case for this book, this method can be used to correct a piece of data manually:

```
UPDATE MeterReading
SET MeterReading = PutElem(MeterReading,
   row('2012-01-01 03:15:00.00000'::datetime year to fraction(5),
   12.0, 1.0, 7.5)::MeterReadingRowType, 0)
WHERE LocID = 12345678;
```

If the location does not exist, then an INSERT must be done first as follows:

```
INSERT INTO MeterReading (LocID, MeterReading)
   VALUES (12345678, TSCreate('ts_15min',
      '2012-01-01 00:00:00.00000',0,0,0,NULL))
```

The two operations can also be combined into one statement:

```
INSERT INTO MeterReading (LocID, MeterReading)
   VALUES (12345678, PutElem(TSCreate('ts_15min',
      '2012-01-01 00:00:00.00000',0,0,0,NULL)
      ::timeseries(MeterReadingRowType),
   row('2012-01-01 03:15:00.00000'::datetime year to fraction(5),
   12.0, 1.0, 7.5)::MeterReadingRowType, 0))
```

For small lists of data, a virtual table interface can be used along with a LOAD statement. For example, if you have to load data for one location for a single day, you can format it as follows:

```
12345678|2012-01-01 00:00:00.00000|11.5|0.2|5.3
12345678|2012-01-01 00:15:00.00000|11.6|0.1|4.2
12345678|2012-01-01 00:30:00.00000|11.9|0.3|1.7
...
12345678|2012-01-01 23:45:00.00000|13.7|0.2|1.2
```

This data can be imported using the following statement:

```
LOAD FROM '/tmp/meter1day.txt' DELIMITER '|'
INSERT INTO MeterReading_v
(LocID, GMT_TimeStamp, UsageTotal, UsageInterval, DemandInterval);
```

This LOAD statement against the VTI table is equivalent to executing either the INSERT or UPDATE statement described previously for each line in the file.

The other problem with these techniques is that the time series is written back to disk for each UPDATE statement executed, meaning this type of update is too I/O intensive.

4.3.2 Large amounts of data

For large amounts of data, a more efficient way to write the data must be used. You can use the following techniques:

- Informix TimeSeries bulkload function
- IBM Informix TimeSeries Plug-in for Data Studio
- Custom code

Informix TimeSeries bulkload function

The Informix TimeSeries bulkload function can load an entire file into a single time series. This is more efficient because the time series is only written to one time, at the end of the file being processed. A sample input file looks like this, with values on the line separated by a tab and ending each line with a new line:

```
2012-01-01 00:00:00.00000	11.5	0.2	5.3
2012-01-01 00:15:00.00000	11.6	0.1	4.2
2012-01-01 00:30:00.00000	11.9	0.3	1.7
...
2012-01-01 23:45:00.00000	13.7	0.2	1.2
```

This data can be loaded using the following statement:

```
UPDATE MeterReading
SET MeterReading =
   bulkload(MeterReading,'/tmp/meter1day_12345678.txt',0)
WHERE LocID = 12345678;
```

The bulkload function will efficiently insert all values from the file into the time series and do the equivalent of a single update at the end of the load.

However, this technique does require organizing the data to be loaded into separate files by time series. For our utility, this would mean creating 1,000,000 separate input files, one per location, along with 1,000,000 separate SQL statements to load these files.

For more information about the bulkload function, see the *Informix TimeSeries Data User's Guide*.

IBM Informix TimeSeries Plug-in for Data Studio

Informix 11.70 xC5 includes a tool for loading time series data. It exists as a plug-in that can be installed into IBM InfoSphere® Data Studio or IBM Optim™ Developer Studio. This solution is a no-coding, efficient method for loading data into time series columns. There are several pieces to the solution.

A *User-Defined Record Format* must be created to define the input file structure. The input file values can be fixed width (for example, columns 1-20) or the values can be delimited. Delimited values can be separated by a single delimiter or two delimiters can surround a value. Delimiters in either case can be one or more characters, and can be different for each field. If the file contains header lines, they can be skipped.

An *Informix Table Definition* must be created for the time series table to be loaded. This definition must include the fields in the row type within the time series.

A *Table Mapping* must be created to define how the user-defined record format fields are mapped to the Informix table definition.

Finally, an *Informix TimeSeries Load Job* must be created to load a file based on the table mapping.

For our utility company records, we can use the same file format as the LOAD example on page 47. In this case, our user-defined record format can be seen in Figure 4-1. Our Informix table definition is shown in Figure 4-2. The table mapping is shown in Figure 4-3. Finally, a load job is shown in Figure 4-4.

There are a few points to note:

- The time series to be loaded must already be created. We can ensure this by simply creating the MeterReading time series when a new location is added to our database.
- The delimiter value is interpreted as a regular expression as defined by **`java.util.regex`**. Because the vertical bar has special meaning (as a logical OR indicator), you have to escape it with the backslash.
- If there are extra fields at the end of the input line that are not needed, these do not have to be defined.

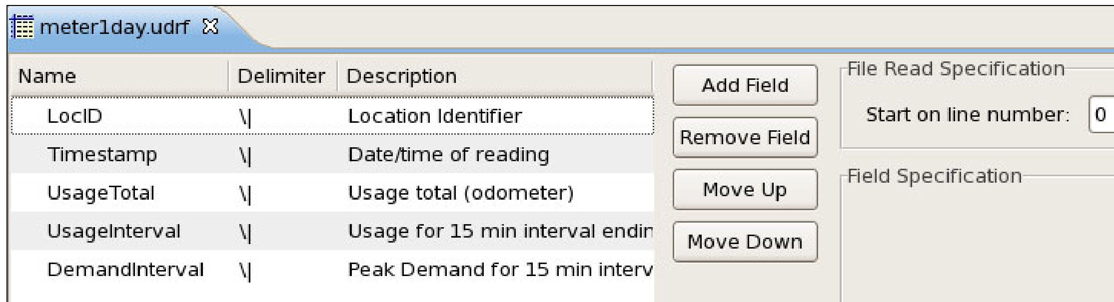


Figure 4-1 User-defined record format example

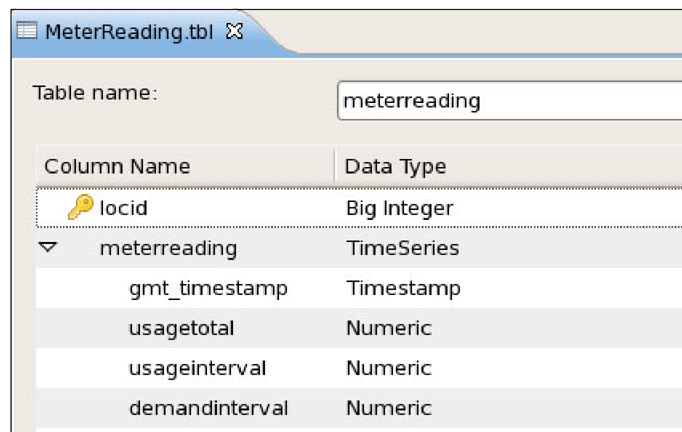


Figure 4-2 Informix table definition example

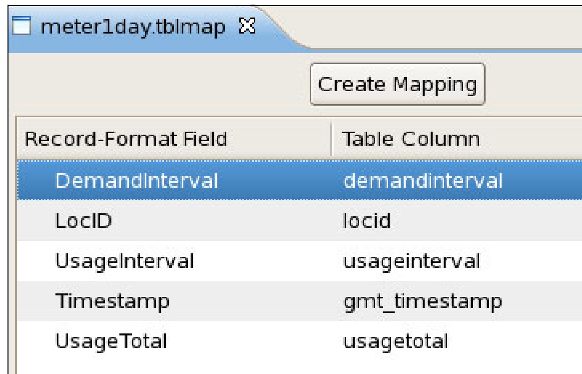


Figure 4-3 Table mapping example

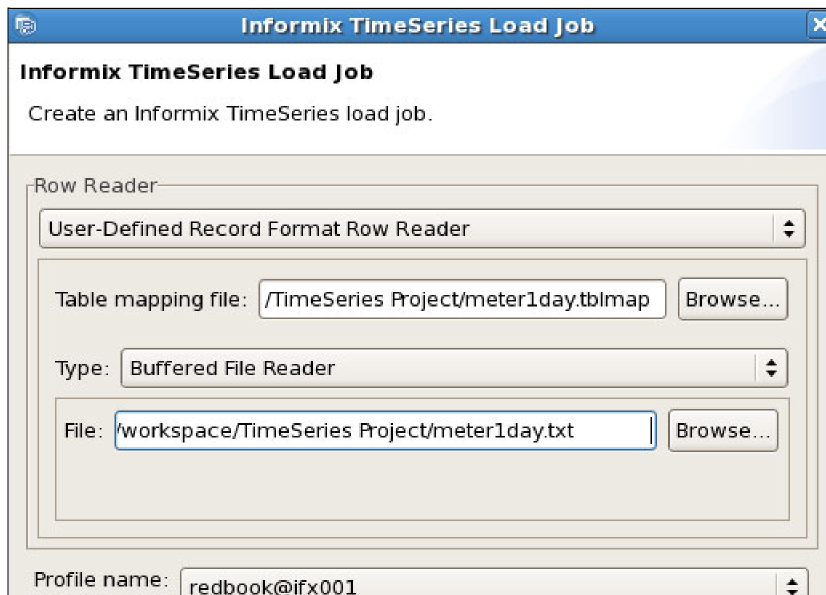


Figure 4-4 Informix TimeSeries Load Job example

Custom code

It is always possible to write custom code to load a time series. The API functions are well documented in the *Informix TimeSeries Data User's Guide*. Within that guide is code for an example custom loader included in Appendix B.

Although this topic is too complex to fully discuss here, keep in mind the following considerations:

- If multithreading is used, two threads should not try to update the same container. Some locking can occur at the container level, which can slow down the writes.
- Although loading from a client connected to the database is possible, it is not as efficient as running on the same machine. There are also some differences in the way the Informix TimeSeries API acts when used on a client machine; these are noted in the API documentation.
- Load all entries for a single time series at the same time if possible. Doing otherwise increases writes and might increase reads.
- Load all entries for a single time series in ascending time stamp order. This load order allows the API to append the new data and is more efficient than inserting the data in some other order, which might cause the existing data to be moved to make room for the new data.

5

Querying TimeSeries data

The scenario for this book stores data in databases so that the data can be retrieved and analyzed easily. This chapter explores several approaches to querying that data, including the advantages and disadvantages of each method. The following approaches are covered:

- Basic relational views
- The TimeSeries SQL API
- Custom functions

This chapter also discusses when to use which approach and gives multiple practical example queries based on the data in the use case described in Chapter 2, “A use case for the Informix TimeSeries feature” on page 15.

© Copyright IBM Corp. 2012. All rights reserved.

5.1 Basic relational views

The Informix TimeSeries feature represents a time series as a data type in the database, a column in a table. Off-the-shelf tools are used to deal with standard relational tables with rows and columns. Informix provides an interface called the *virtual table interface* (VTI) that allows you to create relational views on tables that contain TimeSeries columns. The standard relational columns are part of such a view. If your table includes more than one TimeSeries column, the view operates on only one of them. You will need to create multiple views to cover all the TimeSeries columns.

It is important that you use qualifiers that allow the view to eliminate data, including the selection of TimeSeries and a time interval.

Considering the MeterReading table. You can create a relational view using the following command:

```
EXECUTE PROCEDURE TSCreateVirtualTab('MeterReading_v',
'MeterReading', 'origin(2012-01-01
00:00:00.00000),calendar(ts_15min),container(autopool00000000),
threshold(0),regular', 0, 'Reading');
```

In this command, the following procedure arguments tell Informix what to do if a new TimeSeries must be created:

Argument	Description
`'MeterReading_v'`	Name of the relational view
`'MeterReading'`	Name of the relational table
`'origin(...'`	Definition used for a TimeSeries in a new row
`0`	Flag indicating how to treat use cases (such as null values), which defaults to 0
`'Reading'`	Name of the TimeSeries column (optional if there is only one Timeseries column)

See *IBM Informix TimeSeries Data User's Guide* for more details.

As a result, you get a view that has the following definition:

```
MeterReading_v (
  locid         integer,
  gmt_timestamp  datetime year to fraction(5),
  usagetotal     decimal(15,5),
  usageinterval  decimal(15,5),
  demandinterval decimal(15,5)
);
```

You can use this view to select update and delete values within the TimeSeries contained in the base table.

This particular view can be useful if you need to retrieve the base data from any time series for a specific amount of time. For example, you can retrieve the data for location ID 4727354321171667 for the entire day of 01 Jan 2012 with the following statement:

```
SELECT * FROM MeterReading_v
WHERE locid = 4727354321171667
AND gmt_timestamp BETWEEN '2012-01-01 00:00:00'
           AND     '2012-01-01 23:45:00';
```

What you get as a result can be manipulated by any tools because it is plain relational data.

5.1.1 Expression-based relational views

You might want to take the query a step further. For example, you might not be interested in getting data in 15-minute intervals, but instead, want to aggregate the data on a hourly basis. You can define a view that applies the aggregation internally so that, for you, the view represents data using an hourly interval. This approach allows you to take advantage of TimeSeries-specific functionality while keeping the ease of use of relational tables.

For example, the following command provides an hourly view on top of the base table that contains data at 15-minute interval:

```
EXECUTE PROCEDURE
TSCreateExpressionVirtualTab('MeterReading_hourly_v',
'MeterReading','AggregateBy("min($usagetotal), sum($usageinterval),
avg($demandinterval)", "ts_1hour", Reading, 0)',
'MeterReadingRowType', 0, "Reading");
```

This view automatically applies the AggregateBy SQL function to the selected TimeSeries, making it look like the data was collected at hourly intervals. The table columns are identical to the previous view (`MeterReading_v`). You can then execute the following SQL statement and get one row as a result:

```
SELECT * FROM MeterReading_hourly_v
WHERE locid = 4727354321171667
AND   gmt_timestamp BETWEEN "2012-01-01 01:00:00"
                    AND     "2012-01-01 01:45:00";
```

You can create other expression-based relational views based on your needs. The same concerns apply to this type of views as in the previous case. Make

sure to include a time interval and, if possible, a condition on which meters are being processed.

5.1.2 When to use relational views

You can use relational views, or *virtual tables*, when you know that what you are getting out of the view is exactly what you need to get, without additional processing. For example, you can get the one day of meter data for a specific customer:

```
SELECT * FROM MeterReading_v
WHERE locid = 4727354321171667
AND TSTAMP BETWEEN '2012-01-01 00:00:00'
           AND     '2012-01-01 23:45:00';
```

However, if you want to aggregate values, using relational views is not the most efficient method. For example, you should question the following statement:

```
SELECT locid, SUM(usageinterval)
FROM MeterReading_v
WHERE locid = 4727354321171667
AND gmt_timestamp BETWEEN '2012-01-01 00:00:00'
                  AND     '2012-01-01 23:45:00'
GROUP BY locid;
```

In this case, you are converting the TimeSeries data to relational format and then applying the `SUM` aggregate function. This example is benign because it does not process a large amount of data. In other cases, it can cause the database engine to do additional processing, such as sorting, which is not necessary when using the TimeSeries SQL functions directly.

A quick guideline is that if you are using SQL aggregate functions on a TimeSeries relational view, question this approach and consider changing it. Remember that this approach might work well on a system with a few hundreds of meters and a few days of data, but when it gets to larger volumes, performance problems can increase exponentially.

> **Tip:** A proper approach to the previous statement is to use an expression-based relational view that aggregates the data on a daily calendar.

5.2 Using the TimeSeries SQL API

Informix offers an extensive set of TimeSeries functions to use in SQL statements. When it comes to querying, become familiar with the following function categories:

- Count the number of elements
- Select individual elements
- Extract and use part of a time series
- Find the intersection or union of time series
- Iterator functions
- Aggregate functions

Refer to the *IBM Informix TimeSeries Data User's Guide* for details.

5.2.1 TimeSeries SQL API functions

You will likely use the following set of TimeSeries SQL API functions over and over:

- `AggregateBy`
- `Apply`
- `Clip`
- `GetElem`
- `Transpose`
- `TSSetToList`
- `TSRollup`

Ensure that you understand the capabilities that these functions provide. The examples later in this chapter can help you determine how and where to use several of them.

Relational views are not appropriate for all queries. Some queries require converting too much time series data into virtual tables before performing the operation to eliminate the inapplicable data. Some queries performed through virtual tables first convert large amounts of time series data to relational format and then filter the data. Using the SQL API avoids the overhead of converting data that does not meet the filter criteria.

The use of expression-based relational views allows you to bridge these types of issues because it can execute the TimeSeries SQL functions, which can lead to the creation of multiple expression-based relational views. You have to decide when it is more appropriate to use the API directly. Most likely, it is a matter of flexibility and convenience.

When you use the SQL TimeSeries functions, keep in mind that they operate only on TimeSeries data. This can lead to situations where you want to generate relational representation of the result and you have problems putting the relational data with each TimeSeries element.

For example, say that you want to retrieve one day of data from customers, aggregated by hour, thus getting 24 elements per customer. With the TimeSeries SQL function, you get the following TimeSeries data back:

```
SELECT locid,
       AggregateBy(
"min($usagetotal), sum($usageinterval), avg($demandinterval)",
"ts_1hour", Reading, 0, "2012-01-01 00:00:00",
                        "2012-01-01 23:45:00")
FROM MeterReading;
```

This query returns one row per `locid`. Each row contains a TimeSeries with 24 elements. What can you do if you want to convert this result to the relational format?

TimeSeries provides the `Transpose` function to accomplish this task. In this case, you need to pass the following SQL statement as a string to the function:

```
EXECUTE FUNCTION
transpose("SELECT locid, AggregateBy('min($usagetotal),
sum($usageinterval), avg($demandinterval)', 'ts_1hour',
MeterReading, 0, '2012-01-01 00:00:00', '2012-01-01 23:45:00')
FROM MeterReading",
NULL::row(locid bigint, gmt_timestamp datetime year to fraction(5),
usagetotal decimal(15,5), usageinterval decimal(15,5),
demandinterval decimal(15,5) )
);
```

Important: In this example, the line breaks are there for readability. In reality, the SQL statement is a continuous string.

What comes back is a set of rows of row types based on the definition passed as the second argument (`NULL::row(...)`):

```
ROW(1,'2012-01-01 00:00:00.00000', 150.00000, 40.00000, 10.00000)
```

You can put an SQL wrapper around the transpose execution to get it as multiple columns:

```
SELECT mr.locid, mr.gmt_timestamp, mr.usagetotal, mr.usageinterval,
mr.demandinterval
FROM TABLE (
```

```
transpose("SELECT locid, AggregateBy('min($usagetotal),
sum($usageinterval), avg($demandinterval)', 'ts_1hour',
MeterReading, 0, '2012-01-01 00:00:00', '2012-01-01 23:45:00') FROM
MeterReading",
NULL::row(locid bigint, gmt_timestamp datetime year to fraction(5),
usagetotal decimal(15,5), usageinterval decimal(15,5),
demandinterval decimal(15,5) )
)
) as tab(mr);
```

The key to this statement is to define a table with the `TABLE` keyword and to identify the column name returned as mr with the "`as tab(mr)`" statement. This means that the select statement can reference the row type and extract its columns as shown. The output would be the five columns with the name used in the row type definition of the transpose statement:

```
locid           4727354321000111
gmt_timestamp   2012-01-01 00:00:00.00000
usagetotal      111.09200
usageinterval   0.35100
demandinterval  12.00000
```

This method is the most efficient way to return the results that you want. You can also wrap the statement in a stored procedure to hide the complexity.

If you want to retrieve only the content of a time series, you can also use the `TSSetToList()` function. For example, If you retrieve the data for a specific meter, you might only want the meter data itself:

```
SELECT usagetotal, usageInterval, DemandInterval
FROM TABLE ((
     SELECT TSSetToList(
              ClipCount(MeterReading,
                "2012-01-01 00:00:00", 96, 0
              )
            )::list(MeterReadingRowType NOT NULL)
     FROM MeterReading
     WHERE locid = 4727354321090954
));
```

You can find more practical examples of how to use the TimeSeries SQL functions in 5.4, "Query examples" on page 64.

5.2.2 When to use TimeSeries SQL functions

You use TimeSeries SQL functions when you cannot use relational views. Of course, you can avoid relational views altogether by always using the base table and manipulating the TimeSeries column using these functions.

The question then becomes when *not* to use the TimeSeries SQL functions. Here are some guidelines for your decision:

- Convenience

  If you are satisfied with the performance of relational views, their convenience might compel you to use them instead of TimeSeries SQL functions.

- Complexity

  The use of the TimeSeries SQL functions might make it difficult to write the SQL statements. You can simplify the functions by using a custom function, as described in the next section.

- Performance

  Complexity can lead to performance concerns. In general, if you see yourself nesting TimeSeries SQL functions, you can get a noticeable performance gain by using custom functions. Using the `Apply` function can also lead you to use custom functions, as described in the next section.

5.3 Custom functions

A custom function is written in the C language using the C language TimeSeries API and the server API. It is compiled into a shared library and defined in SQL using the CREATE FUNCTION statement. After it is compiled and defined, the custom function is part of the Informix server and is equivalent to the TimeSeries SQL functions that are provided with the server.

Writing a custom function is not trivial if you are new at it, but after you are familiar with it, it can be done quite quickly. The best way to start is with examples. You can find examples specific to TimeSeries processing in the following locations:

- TimeSeries `examples` directory

  The Informix installation includes a directory of the form `extend/TimeSeries.*/examples`, where the asterisk (*) is a wildcard character that indicates a version number that might change from one Informix release to another.

- IBM developerWorks Informix smart meter central wiki

  https://www.ibm.com/developerworks/mydeveloperworks/wikis/home?lang=en#/wiki/Informix%20smart%20meter%20central/page/Welcome

  This public wiki has a Developers' corner that includes information and example code. It discusses function implementations and allows you to download the code and a makefile. The example code that is provided addresses simple issues that are encountered in real-life situations.

5.3.1 Creating a custom function

After you understand the IBM DataBlade® API and the TimeSeries API, creating a custom function is not difficult; however, using these APIs involves a significant learning curve. Using as many examples as possible is always helpful because examples can get you 90% of the way there. The IBM developerWorks Informix smart meter central wiki mentioned previously includes example code. Specifically, the Developers' corner of that wiki includes example code in a section called "How to create a custom function" that can be useful. This wiki is dynamic and examples are added regularly.

Giving a complete tutorial on how to develop custom functions is beyond the scope of this book. However, the following overview can get you started.

The cntnotnull() function counts the number of elements with a value in a specific column. At the SQL level, the function has the following definition:

```
CREATE FUNCTION CntNotNull(TimeSeries, integer,
                           datetime year to fraction(5),
                           datetime year to fraction(5) )
RETURNING integer
WITH(NOT VARIANT, HANDLESNULLS)
EXTERNAL NAME "$INFORMIXDIR/extend/custom/custom.bld(cntnotnull)"
LANGUAGE C;
```

The function takes the following arguments:

- A TimeSeries
- A column number
- Two datetimes, representing the beginning and ending date

The function returns an integer that represents the number of elements found.

The WITH clause says that the function returns the same result given multiple executions with the same argument, and it does not have side effects, such as modifying a database table. It also says whether the function can still execute if some arguments are NULL. For example if you do not give the dates, it processes

the entire TimeSeries. You can find more information about this function in the *IBM Informix TimeSeries Data User's Guide*.

We then identify the shared library and function name of the implementation. The last line simply says that it is a C function.

The C implementation uses the TimeSeries C API and the datablade API. It must refer to multiple include files:

```
#include "mi.h"
#include "tseries.h"
#include "sqltypes.h"
```

These files are found under the `$INFORMIXDIR/incl/public` directory and in the TimeSeries directory under the `$INFORMIXDIR/extend` directory. The `tseries.h` include file is found in a subdirectory called `lib`.

The function declaration is as follows:

```
mi_integer
cntnotnull(ts_timeseries *ts,
       mi_integer    colno,
       mi_datetime   *start_dt,
       mi_datetime   *end_dt,
       MI_FPARAM     *fParam)
```

The function uses data types defined in the DataBlade API and in the TimeSeries API. The last argument (`fParam`) is an opaque structure that allows us to find information such as whether one of the arguments is a `NULL` value.

The core of the function opens the TimeSeries, positions at the start time and loops over the elements to count the not-NULL values. This function is pretty typical pattern of TimeSeries functions. As mentioned earlier, you can find the code on the IBM developerWorks Informix smart meter central site.

After you write the function, compile it into a shared library. It must include the `tsbeapi.a` library found in the `$INFORMIXDIR/extend/TimeSeries*/lib` directory. After you create the library, make sure to make it available at the path mentioned in the `CREATE FUNCTION` statement.

This section gave you an idea of what is involved in creating a custom function. The best way to do it is to study examples, compile and use them on your development system, and then try writing your own.

5.3.2 When to use custom functions

Why use custom functions? Most of the time, custom functions are chosen for performance reasons. When talking about smart meter data, you are talking about *big data*. In that case, it is more than likely that performance is a major consideration. You might say: “So what? I'll use a bigger system.” In some cases a bigger system will help, but at what cost? Is it better to double the size of the system—and all the related costs, such as hardware, software licences, installation, power requirements, and so forth? Or is it better to spend a few weeks putting together a custom function that might provide 10 times the performance improvement? This situation is not unheard of.

This choice might not always be that clear, but there can be other benefits. A custom function can simplify queries, making them easier to read, write, and understand.

You can also have some specific processing that is somewhat convoluted. A custom function can implement part or all of a proprietary algorithm and centralize the processing in the database server instead of having to implement the code in multiple applications. Also, the reduction of data movement between the server and the application can have a noticeable performance impact.

By writing custom functions, you can make the processing more specific. For example, in a scenario that has 10 million customers, you are likely to be doing billing for about 500,000 customers every day, 20 days out of the month. It is likely that you will want to provide additional services to customers over the web. If each customer accesses your website and performs on average three queries after receiving the bill each month, the system must handle approximately 82,191 queries every day of the year. However, the queries will not be equally distributed throughout the 24 hours of the day. Most queries will occur in a window that can be as large as 18 hours and as small as 4 hours. Most likely, queries will not be distributed evenly throughout the week either. All this processing does not include queries for day-to-day operations. The result is a high transaction rate on a lot of data.

Custom functions allow you to make the processing more specific. Your code does not have to retrieve the description of the TimeSeries element and its data type or types. It already knows. Considering the volume of transactions, that saving in processing can make a difference and allow you to get the best performance out of your system. The end result is delayed upgrades and the related savings that come with it.

5.4 Query examples

The following query examples focus on meter data and illustrate how to get additional information by joining other tables in the query. These examples identify the approaches that make sense, whether a relational view, the TimeSeries SQL API, or a custom function. You can use these examples to develop a feel for what to use where in your own environment.

5.4.1 Getting a day of data for a customer

This example is quite simple and is an example used previously. All the selection is provided in the WHERE clause using the location ID (`locid`) and time boundaries (`gmt_timestamp`). No additional processing is done at the SQL level.

This example is the ideal situation for using the relational view. Here is an example of what the statement looks like:

```
SELECT * FROM MeterReading_v
WHERE  locid = 4727354321171667
AND    gmt_timestamp BETWEEN '2010-11-10 00:00:00'
                     AND     '2010-11-10 00:00:00';
```

Note that the `gmt_timestamp` is returned with a precision to five decimals of a fraction of a second. You might want to cast the result to the desired precision. For example, to return data up to the second, use the following statement:

```
SELECT gmt_timestamp::datetime year to second,
UsageTotal, UsageInterval, DemandInterval
FROM MeterReading_v
WHERE . . .
```

If you do not create a relational view, you have to use the TimeSeries SQL functions. Here is one way to do it:

```
SELECT mr.locid, mr.gmt_timestamp::datetime year to second,
       mr.UsageTotal, mr.UsageInterval, mr.DemandInterval
FROM TABLE (
  Transpose("SELECT locid, ClipCount(MeterReading, '2010-11-10
00:00:00', 96,0) FROM MeterReading
WHERE locid = 4727354321171667",
NULL::row(locid int, gmt_timestamp dtetime year to fraction(5),
          UsageTotal decimal(15,5), UsageInterval decimal(15,5),
          DemandInterval decimal(15,5) )
);
```

Note that the string representing the SQL statement in the Transpose function cannot be split into multiple lines.

5.4.2 Getting a day of aggregated data for a customer

This example uses the same statement used in the previous example. All the selection is provided in the WHERE clause using the location ID (locid) and time boundaries (gmt_timestamp). No additional processing is done at the SQL level. If you have an expression-based relational view that provides the required aggregation, you can use it.

```
SELECT * FROM MeterReading_v
WHERE  locid = 4727354321171667
AND    gmt_timestamp BETWEEN '2010-11-10 00:00:00'
                     AND     '2010-11-10 00:00:00';
```

The use of expression-based relational views becomes mostly a decision of how many relational views you are willing to support over the same data. You will likely determine that you always use the same two or three aggregations. Creating a few more relational views will not impact database administration or the schema understanding of your developers.

Because this example is the same as the previous example and there is an example of how to create an expression-based relational view, you can create the appropriate SQL statements as an exercise.

5.4.3 Comparing two different days for a customer

This example explains how to compare one day with another day from seven days earlier.

The way to write the query depends on what the expected result is. You can decide to return the data in time order and let the application decide how to process the rows for comparison (or execute two separate queries and retrieve the rows in each so they match right away). Another approach is to have one row that shows the values for a particular time and the one for seven days earlier. This example uses the second method.

You can solve this problem using a relational view divided into two separate queries that are then joined together:

```
SELECT t1.locid, t1.gmt_timestamp tstamp1,
       t1.UsageTotal UsageTotal1,
       t1.UsageInterval UsageInterval1,
       t1.DemandInterval DemandInterval1,
```

```
        t2.gmt_timestamp tstamp2,
        t2.UsageTotal UsageTotal2,
        t2.UsageInterval UsageInterval2,
        t2.DemandInterval DemandInterval2
FROM (SELECT * FROM MeterReading_v
            WHERE locid = 4727354321171667
            AND gmt_timestamp BETWEEN '2012-01-17 00:00:00'
                              AND     '2012-01-17 23:45:00'
           ) AS t1,
     (SELECT * FROM MeterReading_v
            WHERE locid = 4727354321171667
            AND gmt_timestamp BETWEEN '2012-01-10 00:00:00'
                              AND     '2012-01-10 23:45:00'
           ) AS t2
WHERE t1.gmt_timestamp =
      t2.gmt_timestamp + interval(7) day to day;
```

Use the TimeSeries SQL functions instead of the relational view. The performance difference should be small, but the statement complexity increases. Look at the previous examples to see the changes that are required.

5.4.4 Get peak usage for a time period, population subset

The peak usage represents the sum of the kilowatt hours (kWh) over the subset of the population. You first need to determine the data you are really looking for:

- The peak 15-minute interval over a week
- The peak hour over a week
- The peak 15-minutes each day over a number of days
- The peak hour over a day

There could be many more possibilities. The peak usage represents the sum of the kWh over the subset of the population but, as shown in the previous aggregation to hour example, it represents an average over the period within one time series.

Be careful when working with a subset of the population. Be mindful of the population size, response-time requirements, and the overall system load. Today, the Informix TimeSeries feature does not take advantage of intra-query parallelism, which is more targeted at OLTP-like systems. This limitation is not usually a problem because the system usually has many users and a large number of queries being executed. The query concurrency takes care of using the system resources to the maximum.

Coming back to the example, if you use the expression-based relational view, you have to do the sum over each time period and then find the highest value. It is better to use the TimeSeries SQL functions in this case. First look at the peak hour in a day.

In this query, you need to aggregate the data on a per hour basis and then sum the population subset. Then, you can find the maximum value. Use the following TimeSeries SQL functions:

- AggregateBy
- TSRollup
- Transpose

Here is a statement for a solution:

```
SELECT FIRST 1 mr.gmt_timestamp, mr.demandinterval
FROM TABLE(
 Transpose(
  (SELECT
    TSRollup(
      AggregateBy('min($usagetotal), sum($usageinterval),
                   avg($demandinterval)',
                  'ts_1hour', MeterReading, 0,
                  '2012-01-01 00:00:00',
                  '2012-01-01 23:45:00'),
     'sum($demandinterval)')::timeseries(
          row(gmt_timestamp datetime year to fraction(5),
              demandinterval decimal(15,5)))
   FROM MeterReading)
 )
) AS tab(mr)
ORDER BY mr.demandinterval DESC
```

Note the following information about this statement:

- The `AggregateBy` line is cut in half for formatting but should not be because it is a string.
- The `Transpose` function does not have the SQL statement between quotation marks, which is the form that takes a TimeSeries as argument. Because `TSRollup` is an aggregate function, it returns only one row that has one TimeSeries column.
- This statement does not include a `WHERE` clause to identify a subset of meters. This would need to be added after “`FROM MeterReading`”.

Because you care only about the `demandinterval` parameter, you can change the `AggregateBy` expression so that it does less work on the two other columns. For

example, you can use `first` instead of `min` and `sum`, which would cut the number of code lines executed over each hour.

The `Transpose` function returns 24 rows, one for each hour of the day. Use SQL to sort the values in descending order and to retrieve only the first row, which will be the highest value.

If you do not need to know the time when the peak occurred, eliminate the `Transpose` and use `AggregateBy` and `GetFirstElem` to retrieve only the value, because the `gmt_timestamp` provides the day:

```
SELECT
  GetFirstElem(
    AggregateBy('max($demandinterval)', 'ts_1day',
      TSRollup(
        AggregateBy('first($usagetotal), first($usageinterval),
avg($demandinterval)',
                    'ts_1hour', MeterReading, 0,
                    '2012-01-01 00:00:00',
                    '2012-01-01 23:45:00'
        ),
      'sum($demandinterval)'
      )::timeseries(
           row(gmt_timestamp datetime year to fraction(5),
               demandinterval decimal(15,5) )
        )
    )
   ).DemandInterval
FROM MeterReading
```

The `AggregateBy` above `TSRollup` allows you to take the maximum value from the 24 elements returned by `TSRollup`. The result is a row from which you extract `demandinterval`. The trick with `AggregateBy` is that you now use a daily calendar and ask for the maximum value. The `gmt_timestamp` is for the beginning of the day. Because you asked for a specific day, you do not need to retrieve it. If you do, the statement has to have the top part of the previous statement where you extract both values from the row.

If you need the time stamp of the peak and you feel that you need better performance than the first statement shown here, you can write your own function, which provides the exact capability required. Compared to the first statement, a custom statement provides the following benefits:

- It eliminates the use of the `Transpose` function to convert 24 elements into rows.
- It eliminates the sorting of the 24 rows.

The function scans through the elements and returns the element that has the highest value. This function is a simple loop over the time series. The return value is not a time series but a row. Assuming that the name of the function is *Peak*, the SQL statement looks as follows:

```
SELECT mr.gmt_timestamp, mr.demandinterval
FROM TABLE(
  SELECT
    Peak(TSRollup(
      AggregateBy('first($usagetotal), first($usageinterval),
                   avg($demandinterval)',
                  'ts_1hour', MeterReading, 0,
                  '2012-01-01 00:00:00',
                  '2012-01-01 23:45:00'),
         'sum($demandinterval)'
         )::timeseries(
              row(gmt_timestamp datetime year to fraction(5),
                  demandinterval decimal(15,5) )
            )
    )
  FROM MeterReading
) AS tab(mr)
```

5.4.5 Billing for a subset of customers

The first thing to consider for the billing is the number of statements that you have to generate and the amount of time that you have to do it. With one million customers, 50,000 statements must be generated each day billing is run, 20 weekdays out of each month. Running one SQL statement to create the 50,000 statements would do them sequentially. If you are running on, say, a 64-CPU system, only one CPU will be busy running billing.

Instead, you might decide to run 50 separate billing statements at the same time in separate sessions. This process still leaves 14 CPUs to do any other work, but now, each session has to process only 1,000 customers. Depending on the overall system performance, you could run your billing up to 50 times faster using this method.

For this example, the 50,000 customers are identified by their billing cycles. Subdivide customers in a billing cycle into multiple parallel jobs to take advantage of all the processors of the system. This could be done by introducing an additional value in the customer record, but it might be possible to use existing values to distribute the load. It could be as simple as using a remainder function (`MOD` in SQL) on the location ID or customer ID. A simple distribution

analysis can tell you if it is appropriate. Even if it is not exactly evenly distributed, it is likely sufficient to take advantage of this approach.

As far as the billing is concerned, the total kilowatt/hour consumption is the sum of all the `UsageInterval` for the desired billing cycle. You cannot use an expression-based relational view for this example because the definition of a monthly cycle does not correspond to the definition of a month and the cycle changes for each day. Instead, use the TimeSeries SQL functions.

You already know the billing dates, so you really want the location ID and the total for the cycle. With that, you can add all the necessary customer information to generate the proper billing statement. Of course, there can be additional information provided, such as usage details on a daily granularity or an average usage over each day at an hourly step. For now, look at getting the location ID and the total for billing.

The SQL statement can be quite simple:

```
SELECT locid,
  AggregateRange(
    'first($usagetotal), sum($usageinterval),
     first($demandinterval)',
    MeterReading, 0, '2012-01-12 00:00:00',
    '2012-02-11 23:45:00'
  ).UsageInterval
FROM MeterReading
WHERE locid IN (4727354321171667, ...);
```

The `AggregateRange` function operates on all the elements within a time range and returns a row type. From there, extract the column that you want and get it in relational format. The result has only the standard relational types, and you get three rows that contain the `locid` and `UsageInterval` in each.

As with the use case description in Chapter 2, “A use case for the Informix TimeSeries feature” on page 15, you will likely require support for different billing programs, such as some flat rates and some tiered rates. These types of programs can complicate a billing statement because you cannot simply sum the usage interval. You will need custom code.

The use case uses an irregular time series to keep track of which billing program occurs when. The rates themselves are kept in a standard relational table. Use the billing program irregular time series to select the appropriate rates and apply them to the meter readings.

The best solution involves custom coding using a stored procedure. You can return one value that is the amount due. You can also return details of the billing.

This example returns a TimeSeries that includes the billing rate and energy consumption for each 15-minute interval. Other possibilities include returning energy consumption and cost and changing the interval to hourly or per billing rate time ranges.

This example also assumes that a billing time range ends at the beginning of the next rate range and that the last rate range provided covers the remainder of the day. Instead, you might use the start and end time that applies to the rate, which provides the option of removing the stop time from the rate range definition.

Our solution requires the definitions of two additional row types:

```
CREATE ROW TYPE TwoValues (
  tstamp datetime year to fraction(5),
  value1  decimal(15,5);
  value2  decimal(15,5);
);
CREATE ROW TYPE details_t (
  segment integer NOT NULL,
  starttime datetime hour to second,
  rate    decimal(15,5)
);
```

Stored procedures can become an important part of your solution implementation. The following procedure covers a large part of what you need to know to manipulate time series data in the Informix Stored Procedure Language (SPL):

```
01 CREATE FUNCTION
02      doBilling(rateTS timeseries(BillingRowType),
03    readings timeseries(meterreadingrowtype),
04    start datetime year to day, stop datetime year to day)
05 RETURNS TimeSeries(TwoValues)
06   DEFINE retval         TimeSeries(TwoValues);
07   DEFINE progID, count  integer;
08   DEFINE curDay         datetime year to fraction(5);
09   DEFINE nextDay        datetime year to fraction(5);
10   DEFINE intervalStep   interval minute to minute;
11   DEFINE cur_t          datetime year to fraction(5);
12   DEFINE t1, t2         datetime year to second;
13   DEFINE r1, r2         decimal(15,5);
14   DEFINE nextStep       BillingRowType;
15   DEFINE currentDetails MultiSet(details_t NOT NULL);
16   DEFINE currentRow     details_t;
17   DEFINE readingRow     MeterReadingRowType;
18   LET retval =
```

```
19        TSCreate(GetCalendarName(readings),
20                start, 0, 0, 0, NULL)::timeseries(TwoValues);
21    LET intervalStep = interval(15) minute to minute;
22    -- Get the current progID
23    LET progID =
24  GetPreviousValid(rateTS, start +
25                    Interval(1) day to day).BillingCodeID;
26    LET curDay = Extend(start, year to fraction(5));
27    -- Get the next program entry
28    LET nextStep = GetNextValid(rateTS, curDay);
29    -- Get the current set of program details
30    LET currentDetails = MultiSet(SELECT segment, starttime, rate
31                                  FROM BillingProgramDetails
32                                  WHERE BillingCodeID = progID
33                                  ORDER BY segment
34                                 );
35    -- Get the active rates that apply to the date range
36    LET readingRow = GetElem(readings, curDay, 0);
37    WHILE (curDay <= stop)
38      IF (nextStep IS NOT NULL) THEN
39        IF (curDay >= nextStep.StartDate_timeStamp) THEN
40          LET progID = nextStep.BillingCodeID;
41          LET currentDetails =
42                     MultiSet(SELECT segment, starttime, rate
43                              FROM BillingProgramDetails
44                              WHERE BillingCodeID = progID
45                              ORDER BY segment
46                             );
47          LET nextStep = GetNextValid(rateTS,
48                                curDay + intervalStep);
49        END IF
50      END IF
51      -- Need to do one day at a time
52      LET nextDay = curDay + interval(1) day to day;
53      LET count = 1;
54      WHILE (curDay < nextDay)
55        FOREACH SELECT starttime, rate INTO t2, r2
56                           FROM TABLE(currentdetails)
57          IF (count <= 1) THEN
58            LET count = count + 1;
59            LET t1 = t2;
60            LET r1 = r2;
61            continue;
62          END IF
63         WHILE (t1 < t2)
```

```
64           LET retval = PutElem(
65               retval, row(curDay, r1,
66                       readingRow.usageInterval)::TwoValues);
67           LET curDay = curDay + intervalStep;
68           LET t1 = t1 + intervalStep;
69           LET readingRow = GetElem(readings, curDay, 0);
70         END WHILE;
71         LET t1 = t2;
72         LET r1 = r2;
73       END FOREACH;
74       LET t2 = "23:59:59"::datetime hour to second;
75       WHILE (t1 < t2)
76         LET retval =
77           PutElem(retval, row(curDay, r1,
78                   readingRow.usageInterval)::TwoValues);
79         LET curDay = curDay + intervalStep;
80         LET t1 = t1 + intervalStep;
81         LET readingRow = GetElem(readings, curDay, 0);
82       END WHILE;
83     END WHILE
84     LET curDay = nextDay;
85     LET readingRow = GetElem(readings, curDay, 0);
86   END WHILE
87   RETURN(retval);
88 END FUNCTION;
```

Lines 1-5 are the function declaration. Note that we use the `CREATE FUNCTION` syntax. It does not really matter when using SPL but the convention is that a procedure does not return anything and a function returns something. In our case, we return a `TimeSeries(TwoValues)`. Also note that SPL does not differentiate between upper-case and lower-case characters. They are used here for readability.

Lines 6-17 are the declarations of variables we use later in the function.

Lines 18-20 create the time series that is returned by the function. The time series starts on the start day of the billing range. This value is received as argument to the function. Note that you must cast the result of the `TSCreate` function to ensure that it returns a time series with the proper sub-type (`TwoValues`).

Lines 23-25 provide the billing rate identification that applies on the start day. Then, line 28 provides the next billing program. Lines 30-34 show the details of the current billing program, which consists of multiple rows identifying the rate and starting time of each range.

The core of the processing is done in the WHILE loop in lines 37-86 that operates on each day between start and stop given as argument to the function. Lines 38-50 show whether you need to get a new billing program.

Another WHILE loop starts on line 54. This is to go over all the time ranges from the billing program. You have to accommodate for the last time range of each billing program, which includes the ones that have only one time range. This is why there is an extra WHILE loop starting at line 75.

The main thrust of the loop is to add each meter reading consumption value (usageInterval) and the rate that applies at that specific time to the TimeSeries. This is done with a statement such as the one on lines 76-78. The rest is about moving to the next reading and looping on the processing.

This function shows you how to create, read, and write time series data and how to use the MultiSet data type to keep track of the billing ranges. This could have been done directly with looping on a SELECT statement but the MultiSet allows us to add some performance optimization because we can re-use the MultiSet without having to go back to reading a table.

6

Managing the ecosystem

This chapter provides information about the methods you can use to administer Informix TimeSeries components. It goes though APIs and tools that can help you manage and view the time series components. Some key performance considerations are also discussed.

This chapter also covers backup, restore, and high availability strategies in the context of the TimeSeries component, and describes methods of interoperability with other third-party products.

© Copyright IBM Corp. 2012. All rights reserved.

6.1 System management and monitoring

You can manage and monitor time series data using many standard Informix techniques. However, some areas, such as the TimeSeries data type, containers, and calendars, require special handling. To simplify this issue, Informix TimeSeries provides more than 100 APIs that can help you manage and query time series data. Although previous chapters have discussed some of these APIs, this section presents a few of the APIs that can help in administering the TimeSeries components.

Apart from the APIs, you can also use the IBM Informix OpenAdmin Tool (OAT) to administer the time series components using the Informix TimeSeries plug-in for OAT. This section provides a walkthrough of the components in the plug-in that are used to administer time series database objects.

6.1.1 Monitor TimeSeries containers using API

Containers are created automatically when they are needed. By default these containers are stored in the same dbspace in which the base table is stored. To store your time series data in other dbspaces, you can create additional containers and move them between container pools. To create additional containers and associate those containers to a named pool, use the TSContainerCreate routine and the TSContainerSetPool.

After you create the containers, monitor the container closely so that you do not run out of storage space. You can monitor the containers for the information about the size and capacity of the time series data storage and also obtain information for a specific container or for all containers in the database.

The sections that follow describe a few of these routines.

TSContainerUsage

This routine returns information about the size and capacity of the specified container or of all containers. This routine is particularly useful for monitoring how full the specified container is, how quickly the containers are filling, and whether you need to allocate additional storage space.

In Example 6-1, the "*mult_container*" container has 26 time series data elements using 30 pages out of the total 50 pages of space. Passing a NULL parameter to the routine returns information for all the containers.

Example 6-1 TSContainerUsage usage

```
$ dbaccess stores_demo -
Database selected.

> EXECUTE FUNCTION TSContainerUsage(NULL);
      pages              slots       total
       2029             241907        2067
1 row(s) retrieved.

> EXECUTE FUNCTION TSContainerUsage("mult_container");
      pages              slots       total
         30                 26          50
1 row(s) retrieved.
```

TSContainerTotalPages

This routine (Example 6-2) returns the number of data pages allocated to the specified container or in all containers. Use this routine to view the size of a container.

Example 6-2 TSContainerTotalPages usage

```
> EXECUTE FUNCTION TSContainerTotalPages(NULL);
      total
       2067
1 row(s) retrieved.

> EXECUTE FUNCTION TSContainerTotalPages("mult_container");
      total
         50
1 row(s) retrieved.
```

TSContainerTotalUsed

This routine (Example 6-3) returns the number of pages containing time series data. You can use this routine to check the amount of data space used and see how full the container is.

Example 6-3 TSContainerTotalUsed usage

```
> EXECUTE FUNCTION TSContainerTotalUsed(NULL);
      pages
       2029
1 row(s) retrieved.

> EXECUTE FUNCTION TSContainerTotalUsed("mult_container");
      pages
         30
1 row(s) retrieved.
```

TSContainerPctUsed

This routine (Example 6-4) is similar to the TSContainerTotalUsed routine except that it tells you the status in percentage.

Example 6-4 TSContainerPctUsed usage

```
> EXECUTE FUNCTION TSContainerPctUsed(NULL);
percent
  98.162
1 row(s) retrieved.

> EXECUTE FUNCTION TSContainerPctUsed("mult_container");
percent
  60.000
1 row(s) retrieved.
```

TSContainerNElems

This routine (Example 6-5) returns the number of time series data elements stored in the container. A time series data element is described as a tuple as defined by the CREATE ROW SQL directive and as such, it encapsulates all the elements within that ROW type.

Example 6-5 TSContainerNElems usage

```
> EXECUTE FUNCTION TSContainerNElems(NULL);
elements
              241907
```

```
> EXECUTE FUNCTION TSContainerNElems("mult_container");
elements
                  26
```

6.1.2 Querying the TSContainerTable system table

The TSContainerTable table is managed by the database server and users do not modify it directly. However, in some circumstances, for example when creating a custom maintenance script, you might need to look at the various container definitions or the association of the script to pools. In such cases, you can query the system tables to get the relevant information.

This type of query can be useful for situations where you need to look at all the currently registered containers in the system to create a new one or to look at the association of containers to the container pools or custom pools, as shown in Example 6-6.

Example 6-6 Querying TSContainerTable system table

```
-- Get all the container names, the associated TimeSeries, storage and
the pool names
> SELECT name, subtype, pool partitiondesc FROM TSContainerTable;
name           raw_container
subtype        meter_data
partitiondesc  raw_container rootdbs 100 50 1049096
pool

name           mult_container
subtype        meter_data
partitiondesc  mult_container rootdbs 100 50 1049095
pool

name           MyContainer
subtype        meter_data
partitiondesc  MyContainer dbs03 2048 1024 -1
pool           autopool

name           MyContainer_2
subtype        meter_data
partitiondesc  MyContainer_2 dbs03 2048 1024 -1
pool           autopool

4 row(s) retrieved.

-- Get all the containers in the default ‘autopool’
> SELECT name FROM TSContainerTable WHERE pool = 'autopool';
name  MyContainer
```

```
name  MyContainer_2

2 row(s) retrieved.
```

6.1.3 Monitoring with IBM Informix OpenAdmin Tool

IBM Informix OpenAdmin Tool (OAT) is a browser administration tool used for managing Informix database servers. OAT provides the ability to monitor and administer one or multiple Informix instances from a single location and when possible provides recommendations for tuning your system through performance data point gathering and analysis.

Getting OAT: OAT comes bundled with the client SDK product bundle (Informix version 11.70 onwards).

Overview of Informix TimeSeries plug-in for OAT

OAT provides the flexibility of easily plugging in your own Hypertext Preprocessor (PHP) based OAT extensions to create functionality that your business demands for administration and monitoring. One such extension is the Informix TimeSeries plug-in for OAT that comes bundled with the base OAT product itself.

You can use OAT with the Informix TimeSeries plug-in for OAT to review and administer database objects that are related to a time series. Some of the key functions of this plug-in are to:

- Review the TimeSeries subtypes, containers, and calendars that are used for the time series data in a database
- Review the tables and indexes that contain TimeSeries subtypes
- Review the columns and virtual tables for tables that contain TimeSeries subtypes
- Monitor the percentage of the space that is used in the containers and in the dbspaces for the containers
- Create and drop containers for data storage

Figure 6-1 on page 81 shows how OAT ties in with the solution architecture for an application.

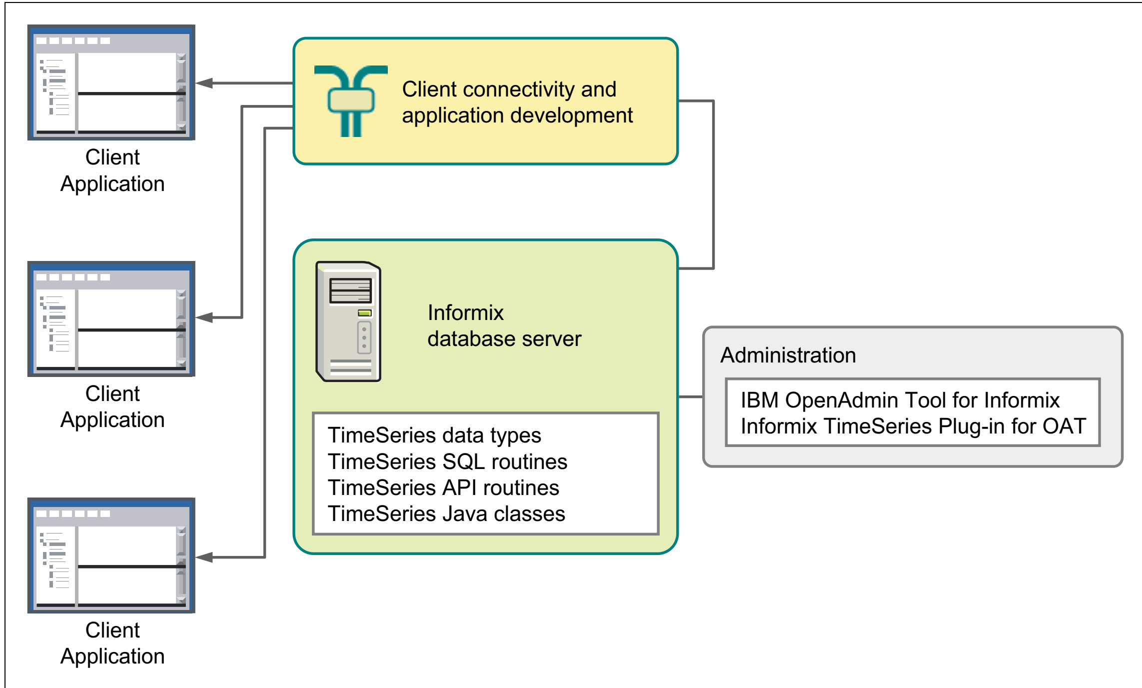


Figure 6-1 Informix TimeSeries OAT administration architecture

Using Informix TimeSeries plug-in for OAT

To open the TimeSeries management console, click **SQL ToolBox** → **TimeSeries**, as shown in Figure 6-2. All the active databases that are associated with the connected server instance display.

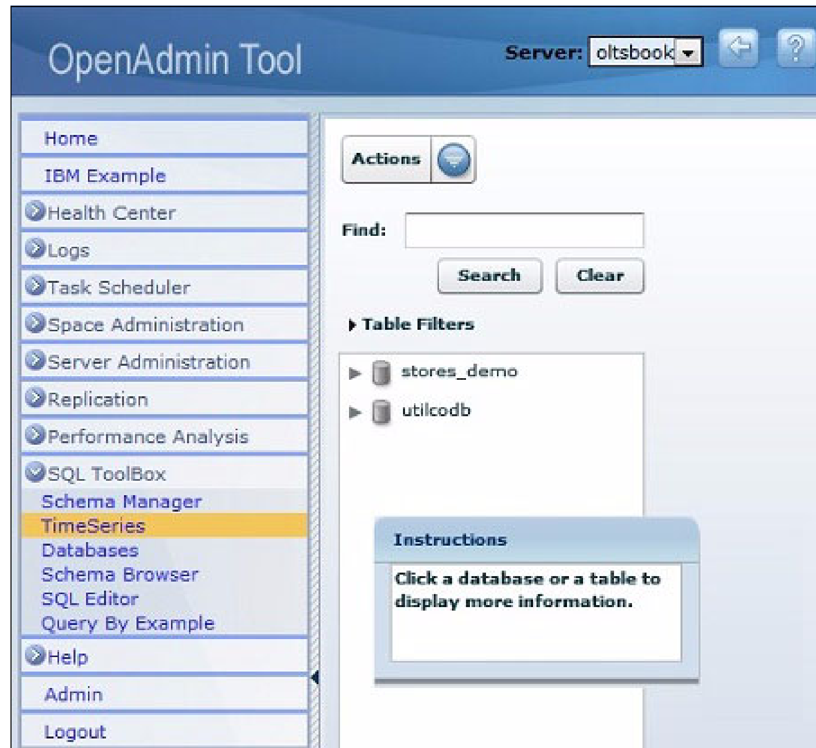


Figure 6-2 Invoking the TimeSeries management console

The TimeSeries management console has a context-sensitive dynamic Actions drop-down menu. Among the options included in this menu are functions such as creating and dropping of containers, calendars, and virtual tables. Each of these options is activated only when you are in the window that contains the context of

the object. For example, when the initial database is selected in the beginning, only the Create Container and Create Calendar options are active, as shown in Figure 6-3.

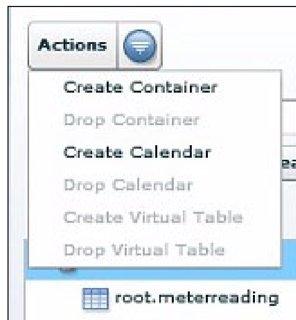


Figure 6-3 Context sensitive Actions menu

How to create these objects is discussed in the later sections.

For now, click the database object name to further review and manage the contents of the database that are related to the time series data types. This dashboard, shown in Figure 6-4, contains information such as database overview, TimeSeries Subtype definitions, Container information, Calendars, and the tables and indexes that are associated with the database.

Figure 6-4 Database drill down

You can expand the information in each of the informational panes by clicking the plus sign (+) at the top-right corner of the pane.

The Information pane gives a generic database overview, as shown in Figure 6-5. Along with various other information, it shows you the logging mode of the database. To store time series data using the Informix data type, the database logging mode must be on. The only other restriction is that the database must not have been created with the 'LOG MODE ANSI' clause selected.

Information

Database name	stores_demo
Owner	root
Created date	2012-03-02
Dbspace	rootdbs
Logging mode	Unbuffered
Locale	en_US.819
Space occupied	12.06 MB
Character case	Case sensitive

Figure 6-5 Database over view

The TimeSeries Subtypes pane lists the row types that have been created to hold time series data (Figure 6-6).

TimeSeries Subtypes

TimeSeries name [ ] Search Clear

TimeSeries	Columns
meter_data	tstamp DATETIME, value DECIMAL
meter_data2	tstamp DATETIME, value DECIMAL, value2 DECIMAL

2 total items 25 Per Page 1 of 1

Figure 6-6 TimeSeries Subtypes

The TimeSeries Subtypes are created using the CREATE ROW TYPE SQL directive. The command definition of the two TimeSeries Subtypes shown in Figure 6-6 can be found in the demo directory of your Informix product installation and is defined as shown in Example 6-7.

Example 6-7 TimeSeries Data Type Definition

```
CREATE ROW TYPE meter_data (
        tstampDATETIME YEAR TO FRACTION(5),
        valueDECIMAL(14,3)
);

CREATE ROW TYPE meter_data2 (
        tstampDATETIME YEAR TO FRACTION(5),
```

```
        valueDECIMAL(14,3),
        value2DECIMAL(14,3)
);
```

The Containers pane (Figure 6-7) displays the details about the storage container and its association to the TimeSeries data type. A container can hold either a regular or irregular time series, but not both together in one container. You can find this information and other details, such as the container space usage and container to container pool association, in this pane.

Containers

Container name [Search] [Clear]

Container	Dbspace	TimeSeries	Regular	Dbspace Used	Contain	Size	Contain
raw_container	rootdbs	meter_data	✓			1.97 KB	
mult_containe	rootdbs	meter_data	✗			50 B	

2 total items 25 Per Page 1 of 1

Figure 6-7 Container details

The Calendars pane lists all the calendars that are defined in the system (Figure 6-8). The calendar list contains the calendar names that come bundled and installed when the TimeSeries data type is initially registered automatically.

Calendars

Calendar name [Search] [Clear]

Calendar	Pattern Start	Pattern	Interval	Number of Time Series
ts_1min	2011-01-01 00:00:00	1 on	Minute	0
ts_15min	2011-01-01 00:00:00	1 on, 14 off	Minute	0
ts_30min	2011-01-01 00:00:00	1 on, 29 off	Minute	0
ts_1hour	2011-01-01 00:00:00	1 on	Hour	0
ts_1day	2011-01-01 00:00:00	1 on	Day	0
ts_1week	2011-01-02 00:00:00	1 on	Week	0
ts_1month	2011-01-01 00:00:00	1 on	Month	0
cal1min	2010-11-10 00:00:00	1 on, 59 off	Second	0
cal15min	2010-11-10 00:00:00	1 on, 14 off	Minute	0
cal1hour	2010-11-10 00:00:00	1 on	Hour	0
cal1day	2010-11-10 00:00:00	1 on	Day	0

11 total items 25 Per Page 1 of 1

Figure 6-8 Container details

Expanding the Tables and Indexes pane (Figure 6-9) provides an overview of the tables and indexes within the selected database.

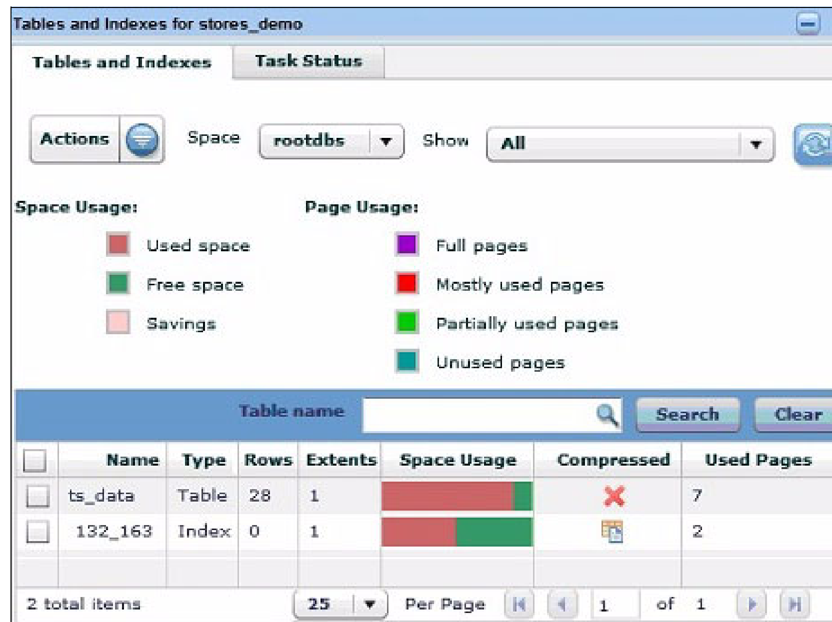


Figure 6-9 Tables and Indexes details

Similar information panes are available when you click a table name and the VTI or Virtual table name in that database.

Tip: The Actions menu associated with each information pane that is related to the Containers, Calendars, and VTI gives the flexibility of adding or dropping those components.

6.1.4 Managing with OAT and APIs

For ease of use, OAT provides an interface through which you can manage the Containers, Calendars, and Virtual tables. You can create and delete these TimeSeries components using the GUI tool the same way you would using the equivalent APIs. However, if you need to do script management of time series data, you must use the APIs.

The following sections demonstrate the process through which each of these components can be managed using either the OAT or the APIs.

Managing containers

To manage containers in OAT:

1. Activate the Create Container option in the Actions menu either by clicking the database name or by expanding the Container window, as shown in Figure 6-3 on page 83. Click **Actions** → **Create Container** to start the Create Container Wizard, as shown in Figure 6-10.

   The wizard prompts are self explanatory.

   Figure 6-10 Create Container Wizard, Pane 1

2. Make the appropriate entries and selections in the Step 1 pane (Figure 6-10), then click **Next**:

   a. Enter the name of the new container that you want to create. This field is mandatory.

   b. Specify the initial container size.

   c. Specify the Size increments value, which is the increment by which the container will grow after the initial allocation becomes full.

Sizing note: The initial default value of 16 KB might be small for applications that need to store a lot of time series data. Depending on your application, and the time series row length and data storage size, enter an appropriate value for your environment. Also, depending on the system and operating system page size, make sure that while allocating space, you allocate enough storage to avoid page splitting between records.

d. Choose the TimeSeries data type that is stored in the container. Notice that the wizard allows only one selection of a TimeSeries data type because a given container can hold data for only one single time series.

e. Specify whether the newly created container will be a part of the default container pool. The default container pool is called the *autopool*. Making the newly created container a part of the autopool is useful when you want to store the time series data in a different dbspace from the table it is in and you do not want to specify the container name when you insert data.

In this case, when you insert data for the time series without specifying a container name, the database server stores the data in the container specified in the screen in the dbspace instead of creating a container in the same dbspace as the table. See Figure 6-11.

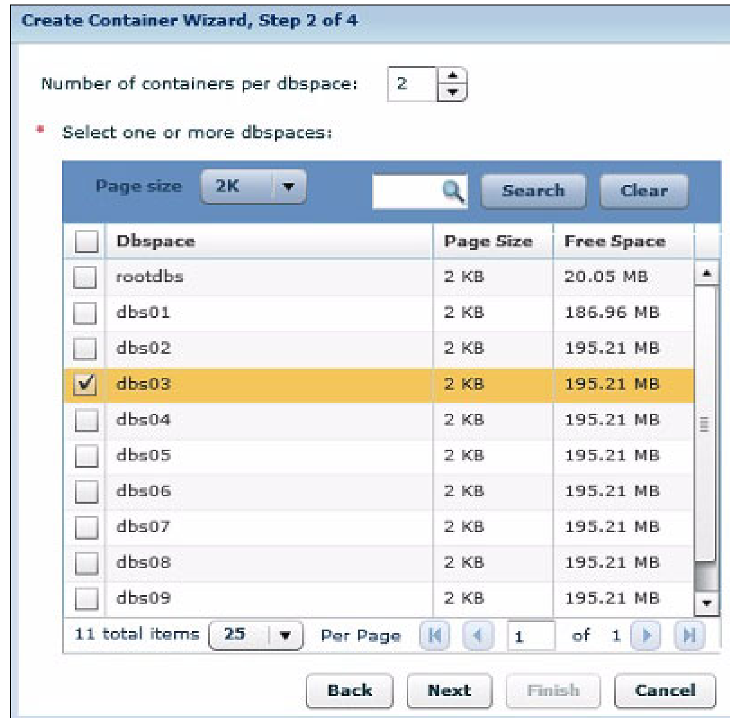


Figure 6-11 Create Container Wizard, Pane 2

3. To create multiple containers with the specified initial settings, select the desired number of containers per dbspace or select more than one dbspace for the containers, as shown in Figure 6-11.

   Click **Next**.

Important: Notice that the dbspace list contains the names of only those dbspaces that are already created and registered in the server. If you need to create additional dbspaces, use the `onspaces` utility to allocate and create a new dbspace before running this OAT wizard.

Also notice that the dbspaces are shown according to page size. If you have created a dbspace in a page size other than the OAT default page size of 2 KB, select the appropriate page size from the drop-down list.

4. The Step 3 pane displays the command that will create the container, as shown in Figure 6-12. Review the command. If there are any issues, click back to the earlier step to make corrections. If everything is correct, click **Finish**.

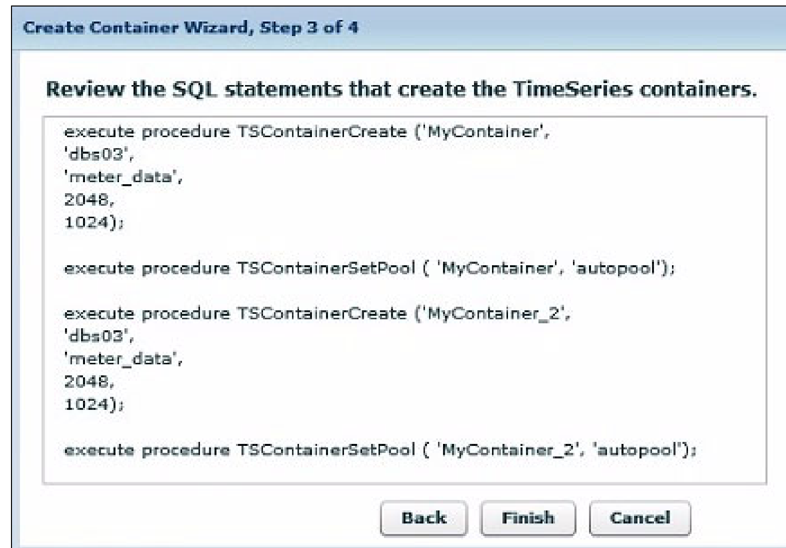


Figure 6-12 Create Container Wizard, Pane 3

5. The specified command is executed on the server, and the outcome of this execution displays in the results window (Figure 6-13). If there are any errors, go back to the previous steps to make appropriate corrections and rerun the command.

Figure 6-13 Create Container Wizard, Step 4 of 4

If you are using the APIs instead of the OAT interface, issue the same commands in `dbaccess` to create the containers. Example 6-8 shows the equivalent execution of the command using the API interface.

Example 6-8 Creating containers in dbaccess

```
-- Create containers
EXECUTE PROCEDURE TSContainerCreate ('MyContainer', 'dbs03',
'meter_data', 2048, 1024);
EXECUTE PROCEDURE TSContainerCreate ('MyContainer_2', 'dbs03',
'meter_data', 2048, 1024);

-- Insert into default container pool
EXECUTE PROCEDURE TSContainerSetPool ( 'MyContainer', 'autopool');
EXECUTE PROCEDURE TSContainerSetPool ( 'MyContainer_2', 'autopool');
```

To drop the container, select the container on the Containers page, and click **Drop Container** on the **Actions** menu.

Managing calendars

The process of creating calendars in OAT is simpler but similar to that described in “Managing containers” on page 88.

To create a calendar, click **Create Calendar** on the **Actions** menu. Enter the pertinent information to create a calendar. The tool allows you to create multiple calendars while in the same window. The Show SQL button displays the command that is executed to create the calendar. If you prefer to use the API instead of the OAT interface, use the command that is displayed by the Show SQL button as a reference to create it using any SQL execution interface.

Figure 6-14 shows an example that creates a calendar named MyWeek. This calendar is defined as three off days followed by four working days. The calendar starts on 2012-06-01, a Friday, which in the context of this calendar would be off on Friday to Sunday and working from Monday to Thursday.

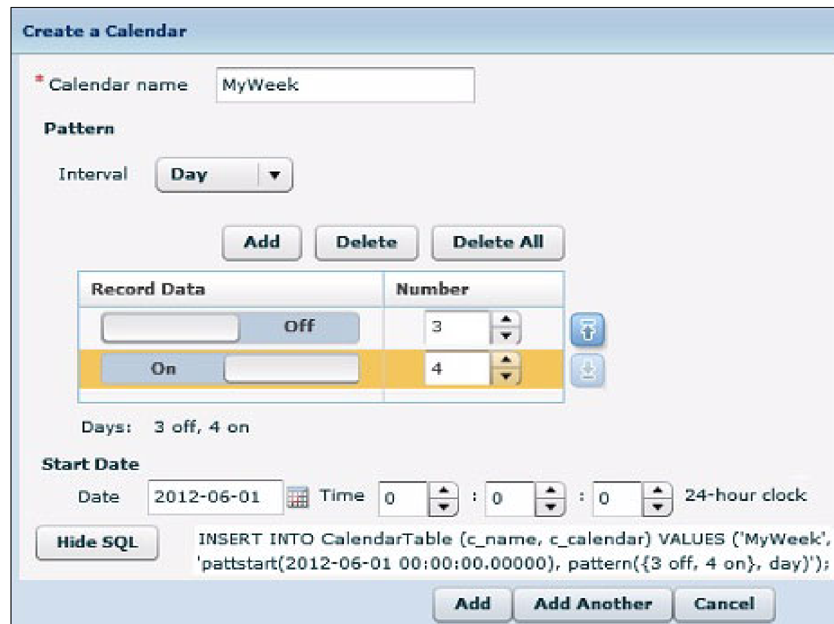


Figure 6-14 Create a Calendar

The equivalent procedure using the APIs as shown in Example 6-9 on page 94.

Example 6-9 Creating calendars using API

```
INSERT INTO CalendarTable (c_name, c_calendar)
VALUES ('MyWeek',
   'pattstart(2012-06-01 00:00:00.00000),
   pattern({3 off, 4 on}, day)');
```

Managing virtual tables

To create virtual tables using OAT, select the table that contains the TimeSeries column for which the VTI is to be created and click **Create Virtual Table** on the **Actions** menu.

Figure 6-15 shows the dialog box to create the VTI.

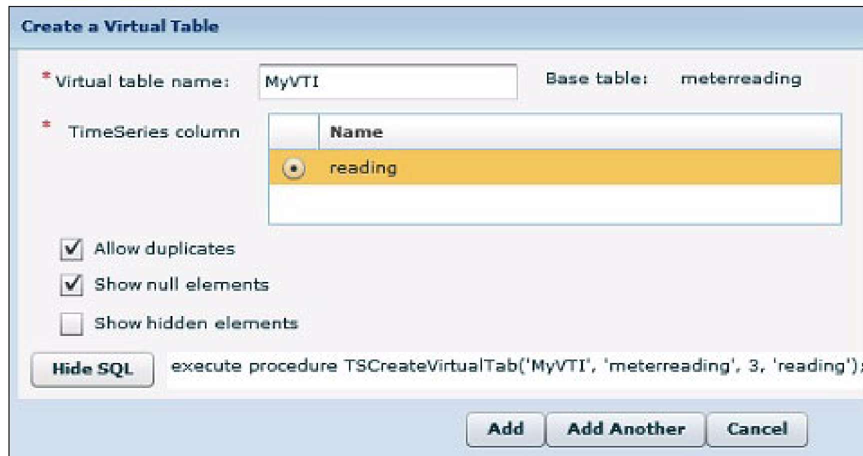


Figure 6-15 Create a Virtual Table

Notice that, even if there are multiple time series columns in a table, only one time series column can be selected for a single VTI. This is by design; only one VTI per TimeSeries column is permitted. You can, however, create multiple VTIs, depending on the number of time series columns that are available in the table. You can also select the VTI properties to define the VTI. You can verify the command using **Show SQL**.

To get a clear understanding of the VTI creation, Example 6-10 shows the components of the base table and the corresponding time series column.

Example 6-10 Base table components

```
-- TimeSeries Row
CREATE ROW TYPE IF NOT EXISTS MeterReadingRowType(
```

```
   GMT_TimeStamp        DATETIME YEAR TO FRACTION(5),
   UsageTotal           DECIMAL(15,5),
   UsageInterval        DECIMAL(15,5),
   DemandInterval       DECIMAL(15,5)
);

-- Base TimeSeries Table
CREATE TABLE MeterReading (
LocID               INT,
Reading             TimeSeries(MeterReadingRowType)
);
```

As of Informix TimeSeries plug-In for OAT v2.75, you can create only the basic VTI using the `TSCreateVirtualTab` routine. To create a virtual tab, based on the results of an expression that was performed on a table containing a TimeSeries column, use the `TSCreateExpressionVirtualTab` procedure shown in Example 6-11.

Example 6-11 Creating virtual tables using API

```
-- Create simple VTI
EXECUTE PROCEDURE TSCreateVirtualTab(
   'MyVTI',
   'meterreading',
   3,
   'reading');

-- Create VTI with expression
EXECUTE PROCEDURE TSCreateExpressionVirtualTab(
   'MyVTIExpr',
   'meterreading',
   'AggregateBy("min($UsageTotal),max($UsageTotal)",
   "cal1min",
   reading)',
   'MeterReadingRowType',
   3,
   'reading');
```

6.2 Performance considerations for TimeSeries data

Unlike the usual relational data, time series data is stored and accessed in a proprietary and specialized way. As such, it requires special considerations for performance tuning purposes. This section covers key points to consider when

handling time series data. These points should be discussed and incorporated beginning with the application design phase. Adhering to and implementing these considerations might help boost performance.

6.2.1 Storage consideration

How you define containers in your hardware storage infrastructure is of utmost importance for achieving high performance. Depending on the type of storage, distribute the dbspaces uniformly across the storage so that one storage location on a spindle is not accessed continuously. Having some dbspaces that are accessed much more often than other dpspaces on the same location can create I/O bottlenecks and thereby degrade performance.

Another factor to keep in mind is to associate one container per dbspace to address the I/O performance bottleneck and to create an environment where space maintenance and administration becomes easy and manageable. Also, to get good I/O across the containers, make sure the key columns, for example meter IDs, are evenly spread within the containers. More containers with fewer key columns will tend to give shorter b-trees (storage structure inside a container) and, thus, faster access.

Furthermore, if you know the total amount of storage required for data retention for a time series row, then try to allocate that amount of space initially in one chunk. This can help to avoid fragmentation of containers when additional space is allocated to extend the container space.

Another design factor that can help boost performance is to create the time series row with fewer columns so that the columns of an element can be contained in a single page. If a single time series data element spans across more than one page, the overhead of fetching the data adds complexity and can also negatively affect performance.

I/O performance is critical, especially when loading data. To minimize the effect of logging updates during data loads, ensure that the physical and logical logs are on different devices and also separate from the container devices. There is no significant I/O to any other parts of the system during the load. Therefore, you do not need to be concerned about rootdbs or having the base table or meter table, for example, on a separate device.

6.2.2 Data distribution statistics

Informix optimizer uses the data distribution statistics stored in the systems table to choose the optimal scan path to fetch the data. Up-to-date statistics are critical for high performance. Make sure that statistics are updated frequently. Informix

scheduler provides one of the many ways by which these system tables can be kept updated automatically. An out-of-date or out-of-sync data distribution statistics system table can severely degrade performance by choosing the wrong query plan.

For Informix TimeSeries, these distributions are kept in the TSInstanceTable system table. To improve performance for any subsequent load, insert, and delete operations, execute the following directives after doing any initial data loading:

```
UPDATE STATISTICS HIGH FOR TABLE tsinstancetable;
UPDATE STATISTICS HIGH FOR TABLE tsinstancetable (id);
```

6.2.3 Memory consideration

Try to size the buffer cache (BUFFERPOOL) in such a way that you can keep the end of the previous day's load and the b-tree pages in cache and have enough space for the current day's load. Ideally, at least one day of data should fit in a single buffer. Use a different page size for all other database objects to separate the TimeSeries buffer pool from the remainder of the system. Typically, the TimeSeries storage uses the smallest page size. All other dbspaces should use the next size larger.

6.2.4 Access consideration

Because Informix TimeSeries stores and maintains row-based data, always create the time series base table with LOCK MODE set to *row*. This setting can help avoid query timeouts compared to when page-level locking is set or during data loads when an exclusive lock on a record is required. It is a preferred practice to use optimistic locking protocols. Another factor to consider is to SET ISOLATION TO DIRTY READ, especially during data insertion to the time series.

6.3 Availability of data

Availability of data, whether it is immediate or archived and backed up for later, is one of the key consideration when designing an application solution.

Informix TimeSeries data does not require any special handling for doing backup and restore. Hence, when deciding on an archiving strategy, the planning should be done in the context of the whole server, not be limited to just the TimeSeries data. Depending on the storage manager, Informix currently provides multiple utilities for backing up and restoring database server data. Because TimeSeries

data is stored in a logical container inside the Informix dbspaces, these utilities back up and restore storage spaces and logical logs that are related to time series just as they do for a non-time series data.

You can also use hardware snapshots provided by the operating system or third-party tools to create a before and after image for archiving and restoring the data.

The data backup using this method is usually off line and is not immediately available for use. To have the data readily and immediately available, either use database mirroring or use replication in the cluster. Currently, TimeSeries data can be replicated using the Informix High Availability Data Replication (HDR) cluster technology.

A high availability cluster consists of two types of database servers:

- The primary database server, which receives updates
- One or more secondary copies of the primary database server

A secondary server is a mirror image of the primary server and is in perpetual recovery mode, applying logical log records from the primary server. If the primary server fails, the secondary server is set to standard mode. The target database connections are redirected to the secondary server and, thus, a continuous availability of data is maintained.

6.4 Interoperability and the complete ecosystem

To be useful, data must be accessible to other software and application solutions from the database in which it is stored. The traditional methods of fetching data between heterogeneous databases and applications are through the ODBC and JDBC interfaces, sometimes using gateways. These methods provide a standard interface for communication between the application and the database.

These methods hold true for Informix as well. Any application can connect and manage the data by connecting to Informix using these industry-standard interfaces.

Although it is simple and straightforward to use the ODBC or JDBC interfaces to connect to Informix data that is stored in relational format, it becomes tricky using the same methodology to fetch the data if it is stored in an Object-Relational format.

The Informix TimeSeries data type is implemented as an Object-Relational architecture and, as such, needs special handling within the database itself. There are many solutions for handling this dilemma and making the

Object-Relational data available to any external application that uses the JDBC/ODBC standards. The sections that follow discuss some of these solutions.

6.4.1 Virtual table interface

The virtual table interface (VTI) is by far the most popular and simple to use strategy for fetching time series data that is stored in an Object-Relational format. The TimeSeries data that is stored internally as an Object-Relational model is converted to a relational view that an application can easily interpret. With VTI, the user does not have to know the underlying proprietary data storage method to get access to the data. Reporting tools, for example the reporters provided in IBM Cognos® application suites, can directly query the virtual tables to fetch the data as needed by the application without having to write custom hooks to interpret Informix TimeSeries data. Business intelligence analytical tools can also populate the facts and dimension tables in their data mart similarly.

Although VTI can be used for querying simple to complex data, there are some restrictions to the way VTI can be used for loading data. You cannot use UPDATE or DELETE statements on time series VTI. You can use SELECT and INSERT statements; however, an INSERT on the VTI table translates to an UPDATE on the underlying time series base table. You can update a time series element in the base table by inserting a new element for the same time point into the VTI.

Due to the ease of use, many applications prefer using VTI as the primary method to load data. One such example is IBM InfoSphere Streams, which through its adapter toolkit (containing ODBC operators) can fetch TimeSeries data from Informix and can insert TimeSeries data into Informix through heterogeneous data sources. This method allows the applications to enhance their capabilities dynamically and perform the relevant analytics with higher precision, improved performance, and reduced storage space requirements.

Although customers prefer using the virtual tables to load data, using a native TimeSeries loading method to directly insert or update data to the time series can provide better performance. Loading data through virtual tables is not as fast as loading data with TimeSeries functions or the Informix TimeSeries plug-in for Data Studio.

6.4.2 TimeSeries APIs

The TimeSeries APIs are specifically designed to work directly with time series data without the need for a middle layer, such as virtual tables. However, not all third-party applications can handle the APIs directly. Some third-party

applications can require an external layer to interpret and manipulate requests and send those requests to the database server. Writing the external layer can be complex, but worth the effort. Using TimeSeries APIs can be an effective strategy for achieving the performance and storage benefits of using the Informix TimeSeries solution.

6.4.3 TimeSeries APIs in Informix stored procedure

Creating a metalayer to use just the TimeSeries API might seem cumbersome to some, especially because it might require an additional maintenance cycle and can add components to the solution. However, the same logic can be coded into the server using the Informix stored procedure routines. The external application or the third-party software then just has to call the stored procedure to manage the data, thus avoiding the metalayer. Instead of creating an external layer to handle the TimeSeries API, you can use Informix stored procedures. If you embed TimeSeries API calls within the stored procedures, the application can call the stored procedures directly without an external layer.

A

Reference material

There are multiple sources of reference material that relate to the Informix TimeSeries environment. They include online documentation, PDF manuals, and an IBM developerWorks wiki.

The subjects cover system management, SQL syntax, TimeSeries utilization, Informix extensibility information, and more.

To begin with, you should know how to install and manage the system. Then you should learn about the general SQL language. When this is covered, you can start using TimeSeries and then consider extending the capabilities if you so desire.

© Copyright IBM Corp. 2012. All rights reserved.

A.1 Online documentation

You can find the online documentation for Informix Version 11.7 at:

http://publib.boulder.ibm.com/infocenter/idshelp/v117/index.jsp

Note that new releases of the product come out regularly. To make sure you are using the latest release of the Informix documentation, go to:

http://www-01.ibm.com/software/data/informix/library.html

Select the **Information centers** link and then select the appropriate **View page** link for the information center that you want to visit.

A.2 The IBM developerWorks wiki

There is wiki called *Informix smart meter central*. Although the focus of the wiki is smart meters, the information here is still valid for any user of Informix TimeSeries.

https://www.ibm.com/developerworks/mydeveloperworks/wikis/home?lang=en#/wiki/Informix%20smart%20meter%20central/page/Creating%20and%20loading

The main sections include:

- News
- Informix videos
- Ecosystem
- Collaterals
- Developers' corner

The Developers' corner includes basics such as how to create a table with TimeSeries, how to load data and how to access TimeSeries. It also includes examples of how to write custom functions.

A.3 PDF manuals

The PDF versions of the Informix manuals are available for free download at:

http://www-304.ibm.com/support/docview.wss?uid=swg27019520

You will find a table listing all the manuals and the languages they are available in. Note also that manuals change with each version of the product. You can find links to the PDF manuals in the home page of the info centers.

It can be overwhelming to see such a long list of manuals. Here is a short list of the manuals you will want to consult.

- System administration
  - *Administrator's Guide*
  - *Administrator's Reference*
- SQL
  - *Guide to SQL: Reference*
  - *Guide to SQL: Syntax*
  - *Guide to SQL: Tutorial*
- TimeSeries
  - *TimeSeries Data User's Guide*
- Extensibility
  - *DataBlade API Programmer's Guide*
  - *DataBlade API Function Reference*

B

Enterprise historian database example

This appendix elaborates on the advantages of the Informix TimeSeries approach over standard relational storage with the help of popular solutions in the energy sector. The example concentrates on size of data, application development, and performance and interoperability with other products.

Achieving operational excellence requires you to collect and optimize vast data from across operations for true process visualization. Providing business context to the same operational data by connecting islands of information enhances the analytical capabilities and facilitates better and faster business decisions at the enterprise level.

The typical Enterprise Historian collects data from operational historians or operational real-time databases and creates a common repository with other business contextual information coming from customer relationship management (CRM), enterprise resource planning (ERP), and so forth, thereby developing a comprehensive analytical model for the business.

The major chunk of data in such a system is always operational data, which often is time series data. Due to the large size, time series data imposes maintenance, storage, performance, application development, and data integration challenges. There are quite a lot of technologies that address performance and storage challenges but that lack in extendability, generality, and interoperability.

© Copyright IBM Corp. 2012. All rights reserved.

This appendix discusses how the Informix TimeSeries technology addresses these challenges with the help of use cases in a Wind Power Generation Historian.

B.1 Wind power generation historian

Typically, a wind power generation historian needs to collect the following operational and incidental data for each turbine and perform analytics:

- Turbine utilization
- Wind versus power generation trends
- Wind rose analysis
- Average wind speed versus power generation trends at different aggregation level (hourly, daily)
- Turbine downtime and roll up to farm level

B.2 Disk space savings

Typically, for handling 600 tags per turbine, with 30 turbines per site, at 1-minute intervals for data, and a data retention period of 36 months, the data size of time series data for one site grows to 1.66 TB in a standard relational database management system (RDBMS) approach. However, when you store the same data using Informix TimeSeries technology, the data size is reduced to 15%, or 224 GB, for space savings of almost 85%.

Scaling this to 20 sites makes it unmanageable with a standard RDBMS approach because the data grows to 32 TB. However, with the Informix TimeSeries approach, it grows to a maximum of only 5 TB.

Standard RDBMS approach

In a standard relational approach, create the table for storing operational information as shown here:

```
create table farm_data (
loc_id       integer, -- 4 bytes
plant_id     integer, -- 4 bytes)
tag_id integer -- 4 bytes
timestamp     datetime year to fraction(5), -- 11   byte
value decimal(7,2) -- 5 bytes
)
```

Then, create the composite index on the `loc_id`, `plant_id`, `tag_id`, and `timestamp` columns as follows:

```
create index ind_01 on farm_data(loc_id,plant_id,tag_id,timestamp)
```

To store the data for 18,000 tags per minute for 36 months, the required storage calculation is as follows:

- Record size = 4 + 4 + 4 + 11 + 5 = 28
- Record/slot overhead = 4
- Total size of a record = 32 bytes
- Page overhead for maintaining page information = 28 bytes per page
- Total free space on 2 KB page size = 2048 - 28 = 2020 byte
- Total records on each page = 2020/32 = 63
- Total number of records = 18000*60*24*30*36 = 28 billion
- Total number of pages used = 28 billion/63 = 444 million
- Total space required for data = 444 million * 2 KB = 847 GB

Thus, the total space that is required for the index is 820 GB.

And, the total space that is required is 847 + 820 = 1667 GB = 1.66 TB.

Informix TimeSeries data type approach

With the Informix TimeSeries approach, create a row type named `row1`, and create the table named `farm_data_ts`, which contains a time series column as follows:

```
create row type row1 (
timestamp datetime year to fraction(5),
value decimal(7,2)
);

create table farm_data_ts (
  loc_id       integer, (4 bytes)
  plant_id     integer, (4 bytes)
  tag_idinteger (4 bytes )
  operational_data timeseries(row1)
)
```

Then, create the composite index on the `loc_id`, `plant_id`, `tag_id`, and `timestamp` columns as follows:

```
create index indx_1 on farm_data(loc_id,plant_id,tag_id)
```

To store the data for 18,000 tags per minute for 36 months, the required storage calculation is as follows:

- Store the `loc_id`, `plant_id`, and `tag_id` only once. Thus, the space that is required these columns is 18000*12 = 211 KB.
- Create the index on `loc_id`, `plant_id`, and `tag_id` columns. Thus, the total space that is required for the index is 18000*36 = 281 KB.

The space that is required for all remaining columns is as follows:

- Record size = 5
- Record/slot overhead = 4 bytes
- Total size of record = 9 bytes
- TimeSeries index = 4+4+4+8+4 = 24 bytes
- Page overhead for maintaining page information = 28 table overhead Total free space on 2 KB page size = 2048 - 28 = 2020 bytes
- Total records on each page = 2004/9 = 224
- Total Number of records = 18000*60*24*30*36 = 28 billion
- Per day index size = 126000 * 24 = 2.5 MB
- Total pages required for data = 28 K million/224 = 238 GB

Thus, the total space that is required for an index of 36 months is 2.7 GB.

And, the total space that is required is 238 GB + 211 KB + 281 KB + 2.7 GB = 241 GB.

Figure B-1 illustrates the space savings.

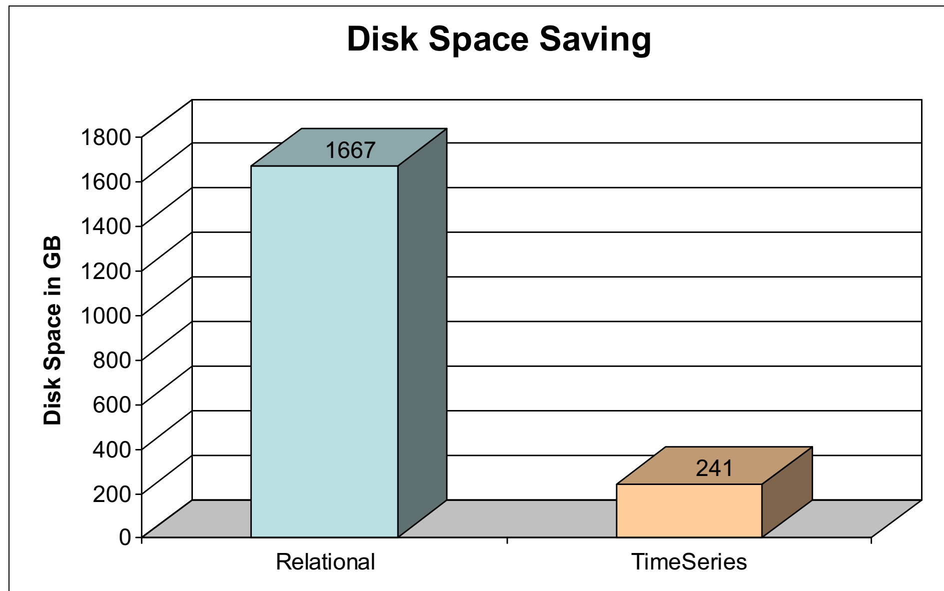


Figure B-1 Space savings

B.3 Application development and performance

Turbine utilization for a specific period is the ratio of expected power generated per hour to actual power generated per hour by the turbine in that period. Assuming the tag named `generated_kwh` with `tag_id 1001` stores the incremental value, the turbine utilization query is as shown in Example B-1.

Example B-1 Turbine utilization query

```
SELECT loc_no,plant_no,
(
 (
  (row2.value - row1.value)/ time_diff_hour("2011-11-01
   00:00:00","2011-11-30 23:59:59")
  )*100
)/800 p_g_per_hour
FROM
 (
    SELECT loc_no,plant_no,
    getnextvalid(operational_data,
       "2011-11-01 00:00:00"::datetime year to second) row1,
```

```
    getpreviousvalid(operational_data,
        "2011-11-30 23:59:59"::datetime year to second) row2
    FROM farm_data
    WHERE tag_id=1004 and loc_id=3091 and tag_id=1001
)
```

In Example B-1:

`getnextvalid`	A built-in routine returns the nearest entry after a specified time stamp
`getpreviousvalid`	A built-in routine returns the nearest entry before a specified time stamp
`time_diff_hour`	A simple user-developed function that calculates the time difference in hours between two time stamps
800	Expected kilowatts per hour

Figure B-2 shows the result.

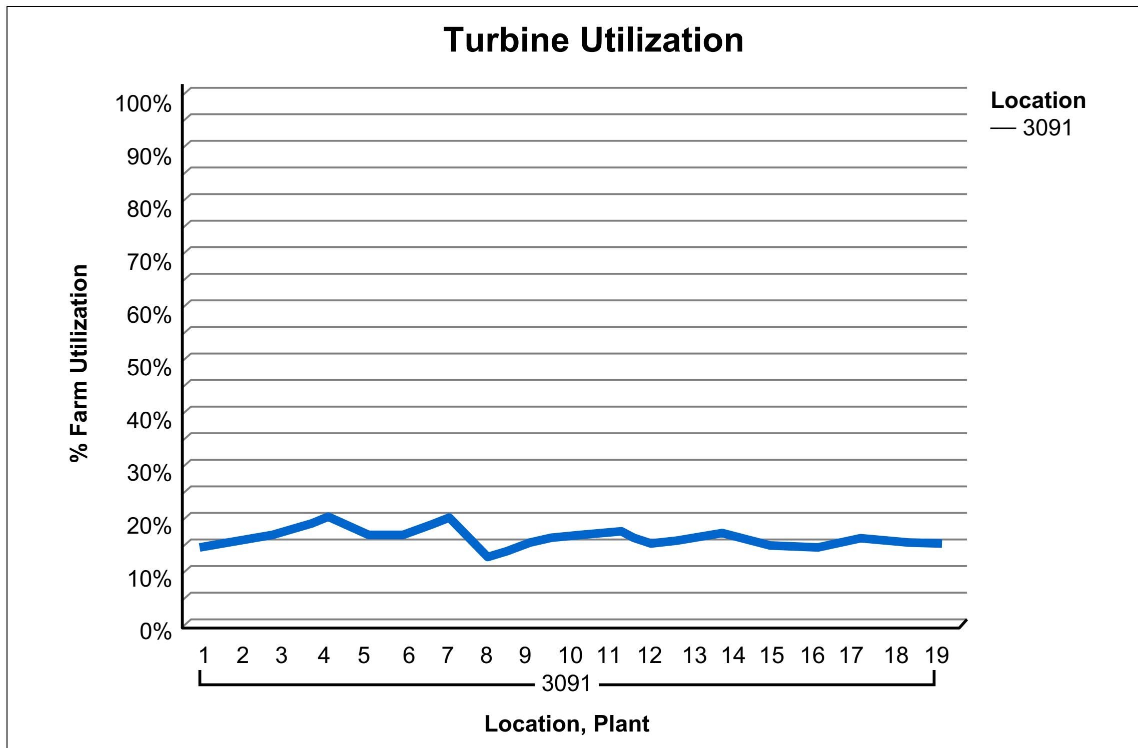


Figure B-2 Turbine utilization chart

Because Informix does offset-based calculations to reach out to the required record, fetching the first and the last time series element in the specified period, regardless of the time range in between, is fast. However, a similar operation in a traditional relational approach is highly resource intensive, where one needs to read complete data in the particular range and perform an "order by" clause on it.

Another frequently asked requirement of the historian solution is an aggregation of data at different levels, such as hourly and daily. A report, such as the Average Wind Speed versus Power Generated report, needs aggregation at a different level. This aggregation can be done easily with a simple SQL query using the built-in TimeSeries functions, such as the `aggregateby` function shown in Example B-2. This example assumes `1002` is a `tag_id` for wind speed and `1003` is a `tag_id` for the power generated.

Example B-2 The aggregateby function

```
SELECT * FROM table((
 SELECT
 tssettolist
 (
  (
   aggregateby
     ('avg($value)',
      'ts_1hour',
      operational_data
     )::timeseries(one_decimal)
  )
 )::list(one_decimal not null)
 FROM farm_data
 WHERE loc_no=3091 AND plant_no=1 AND tag_id=1002
));
```

Figure B-3 shows the result.

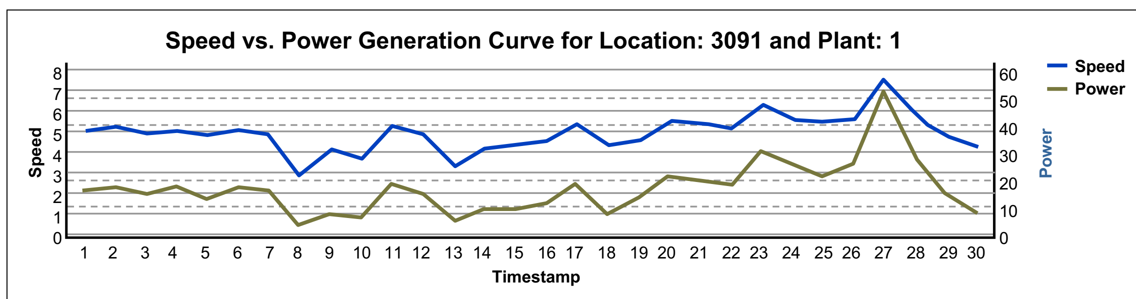


Figure B-3 Historian speed versus power

This query gives the average wind speed per hour. The calendar passed to the aggregateby function, ts_1hour, follows an hourly pattern. The aggregation can be changed to any level just by changing calendar pattern as shown in Example B-3.

Example B-3 Average wind speed per hour query

```
SELECT * from table((
 SELECT
 tsSetToList
 (
  (
   aggregateby
     ('avg($value)',
      'ts_1hour',
      operational_data
     )::timeseries(one_decimal)
  )
 )::list(one_decimal not null)
 FROM farm_data
 WHERE loc_no=3091 AND plant_no=1 AND tag_id=1003
));
```

In Example B-3:

ts_1hour	Calendar following an hourly pattern
aggregateby	A built-in routine that returns aggregate value aggregation condition at the level defined by the calendar
one_decimal	A row type having time stamp and a decimal (7,2) value
tsSetToList	The TSSetToList function takes a TimeSeries column and returns a list
(collection of rows)	Containing all the elements in the time series

Similarly, the Wind Rose Analysis can be done easily by looking at data aggregated to a different level. The three values that you need to pass to the wind rose chart are average wind speed, time stamp, and wind direction, as shown in Figure B-4.

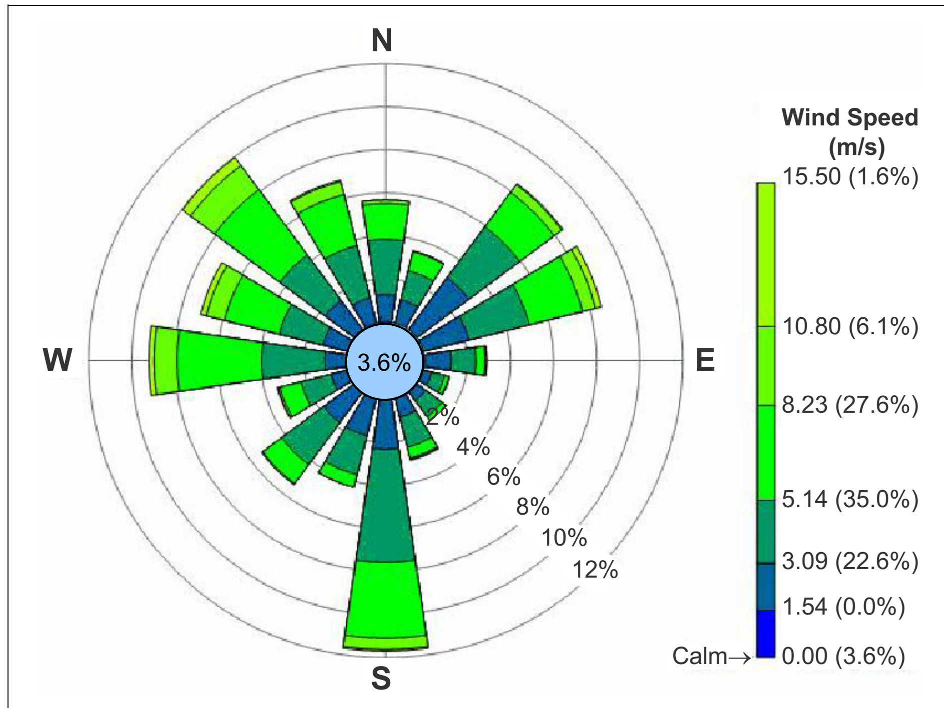


Figure B-4 Wind rose analysis

Assume that `1004` is the `tag_id` that stores the nacelle position details. The following function brings any value in the range of 0 to 360:

```
CREATE FUNCTION between_0_360(position decimal(7,2))
RETURNING decimal(7,2);
DEFINE new_pos decimal(7,2);

LET new_pos = position;

IF new_pos < 0
THEN
   WHILE new_pos < 0
      LET new_pos = new_pos + 360;
   END WHILE;
ELSE
   WHILE new_pos > 360
```

```
        LET new_pos = new_pos - 360;
     END WHILE;
  END IF;
  RETURN new_pos;
  END FUNCTION;
```

The following function brings the value of nacelle position in the range of 0 to 360:

```
CREATE FUNCTION ts_conv_360(a row1)
RETURNS one_decimal;
RETURN row(null::datetime year to fraction(5),
between_0_360(a.value))::one_decimal;
END FUNCTION;
```

The following SQL query gives the aggregated value of nacelle position at day level for `loc_id` 3091 and `plant_id` 1:

```
SELECT * from table ((
SELECT tssettolist ((
aggregateby('avg($value)','ts_1day',ts1)
   )::timeseries(one_decimal)
    )::list(one_decimal not null)
   FROM
   (
  SELECT apply('ts_conv_360',
  "2011-11-01 00:00:00.00000"::datetime year to fraction(5),
  "2011-11-30 23:59:00.00000"::datetime year to fraction(5),
   operational_data
   )::timeseries(one_decimal) ts1
FROM farm_data
WHERE loc_id=3091 AND plant_id=1 AND tag_id=1006 )));
```

The following SQL query gives the aggregate value of wind speed at day level for `loc_id 3091` and `plant_id 1`:

```
SELECT * from table((
 SELECT
 tsSetToList
 (
  (
   aggregateby
     ('avg($value)',
      'ts_1day',
      operational_data
     )::timeseries(one_decimal)
  )
```

```
  )::list(one_decimal not null)
  FROM farm_data
  WHERE loc_no=3091 AND plant_no=1 AND tag_id=1003
));
```

B.4 Interoperability

One of the major roles that historian plays is to integrate operational information with other business contextual information. And having easy interoperability with third-party tools makes it easy to build multiple solution blocks. Most of the non-RDBMS historian solutions compel the use of specialized client adaptors like OPC and modbus to fetch data from historian. The result is that the solution looses interoperability and generality.

Because TimeSeries is a feature of Informix, it allows any third-party tool or application development tool to access data from historian using standard database drivers, such as ODBC, JDBC, PDO, and so on. Thus, for developing monitoring interfaces or reporting layers on top of Informix-based historian, the solution architect has a vast choice of tools, such as Jviews, simple Java, Cognos, SAP MII, and PHP to develop monitoring interfaces and reporting layers.

B.5 Summary

The Informix TimeSeries approach is a breakthrough technology for managing and analyzing time series data in solutions like historian. This approach offers the distinct advantages of huge savings on disk storage, extremely high performance, and ease of application development. The built-in SQL routines and extensions of C APIs and JAVA APIs allow users to develop and incorporate their own program logic into the database engine faster. Because it is easy to learn and use, Informix is widely accepted by the software community as meeting time series data management requirements, compared with other database and non-RDBMS solutions.

C

Distribution grid monitoring enabler

This appendix describes a distribution grid monitoring enabler solution used for complex real-time data acquisition and monitoring.

Utilities today have limited access to load data and fault location information in the distribution segment of an electric network. The following practices are typical present day MV and LV network monitoring practices:

- Manually-read earth or line fault indicators are installed as standard practice on 11kV/415V kiosk (padmount) substations to indicate to field operators whether a fault current has passed.
- In the medium/low voltage network, maximum demand indicators can be installed at most kiosk substations and are read manually (on an annual or more frequent cycle) to record the highest substation load since the last device reset.

Alternatively, if neither of these practices is in place, the utility relies on estimated load information. This estimation is based on high-level supervisory control and data acquisition (SCADA) information, standard load profiles, and online patrols to find MV faults.

Outage recording is largely manual and requires lengthy and manual data matching and analysis. Fault identification and fault location information also rely

© Copyright IBM Corp. 2012. All rights reserved.

on field-based manual assessment and the tacit knowledge of control room and field-based crews built up over years. Load data required for larger customer supply connections, network reliability assessment, augmentation and replacement is typically collected based on manual field-based processes.

C.1 Solution overview

The distribution grid monitoring enabler solution is the joint solution from IBM and PowerSense. Although PowerSense provides the smart sensors that are required to monitor the grid, IBM provides the platform that acquires, stores, and manages this data.

The distribution grid monitoring enabler gives context to the data and makes this data accessible through a visual interface and a programmatic CIM-based interface. This offers analytics, such as unbalanced feeders and hot substation utility operations, without modification and planning insights. It also offers a user interface to manage devices, such as firmware and settings, remotely.

Figure C-1 on page 119 shows the architecture.

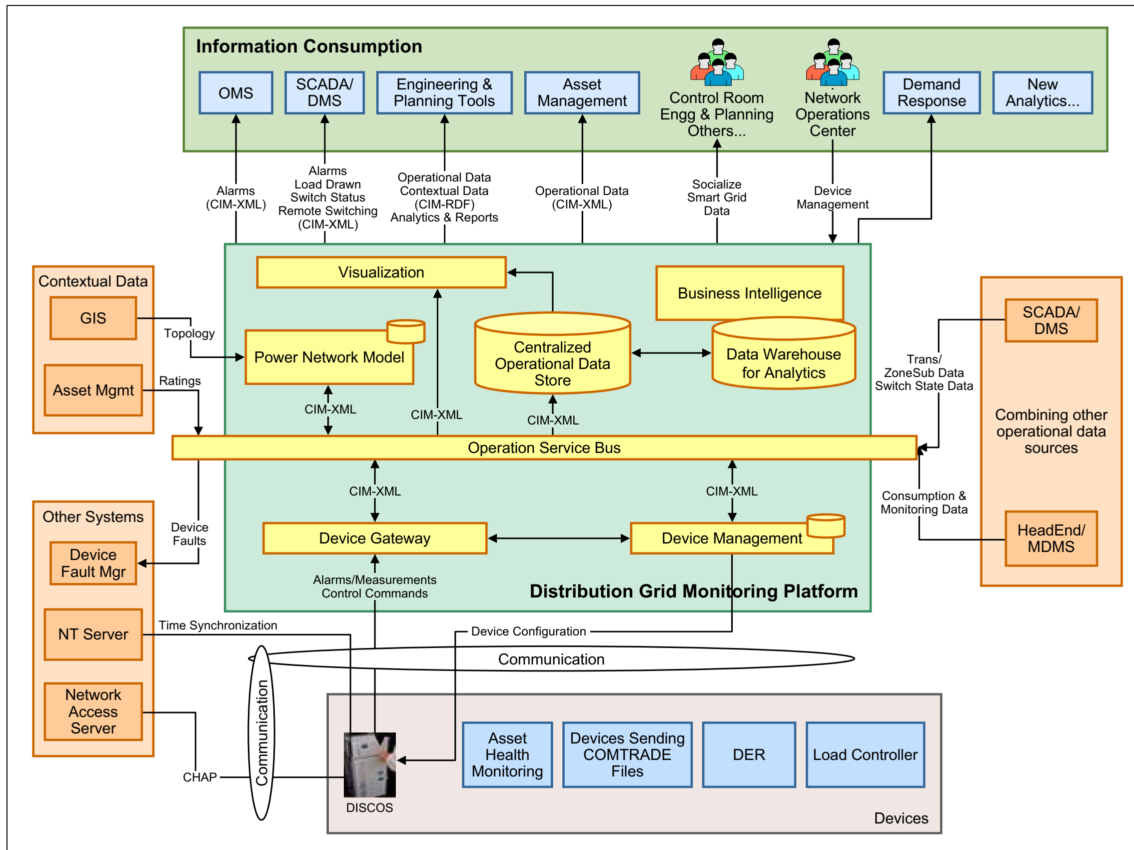


Figure C-1 Distribution grid monitoring enabler architecture

C.2 Business benefits

The distribution grid monitoring enabler solution offers the following business benefits:

- Improved time to diagnose and rectify outages:
  - Proactive reporting of faults instead of waiting for customer to report fault
  - Remote diagnosis of fault location on 11kV feeder
  - Remote setting of earth fault indicators
- Avoided costs of dealing with manual instrument readings:
  - Maximum demand indicators no longer must be read every 6 months.
  - Load surveys can be avoided due to the presence of accurate measurement data.

- Improved network planning and management of capital expenditures:
  - More data is available to support planning and forecasting.
  - Network Augmentation Deferrals.
  - Power Factor and Phase balance correction.
- Outage data management
- Reduce battery monitoring costs
- Faster decision on new connections (customer > 100 amps)
- Proactive customer voltage investigations
- Inbound outage call management

C.3 Challenges

Generally, in a full blown implementation, the distribution grid monitoring enabler solution needs to handle over 3.5 million tags that represent the network points being monitored in a distribution grid. The operational data about all these tags must be acquired real time, and real-time dashboards are monitored for grid health checkup. The sensor devices potentially can be configured to transmit information every 10 minutes. Hence, the required data insertion rate is a minimum of 10,000 records per second and the database should be able to perform read operations as well as database administration activities with high efficiency in parallel.

Considering 3.5 million tags and a data retention period of 18 months, the data size with traditional RDBMS grows up to 26 TB. To address the performance and data size challenges, initially the non-RDBMS solutions were preferred. However, in a system such as the distribution grid monitoring enabler solution, although operational data is one of the major factors to determine the technology, other components such as Power Network Model (PNM) also must be considered. The ultimate output of the system is achieved by establishing relation between PNM and operational data. The PNM is purely a relational model and cannot be handled with non-RDBMS solutions. Thus, having operational data in non-RDBMS system and PNM in RDBMS adds another challenge of integrating these systems to achieve final output.

C.4 Solution

Informix not only solves all the challenges but also makes it easy to extend the capabilities of the distribution grid monitoring enabler solution by adding SCADA, master data management (MDM), and warehouse capabilities. Informix TimeSeries takes care of performance and keeps data size under control. The general RDBMS capabilities handle PNM efficiently. Because the modules now reside in Informix, it is easier to integrate modules and establish relations between operational data and PNM.

The operational data is stored in simple TimeSeries table name measurements:

```
CREATE ROW TYPE measurement_row
(
timestamp datetime year to fraction(5),
value decimal(7,3)
);

CREATE TABLE measurements
(
msmt_tag  varchar(250),
msmt_type varchar(20),
measurement timeseries(measurement_row)
);
```

The measurement tag ID is the unique identification number of the device, derived from the PNM module stored in the `msmt_tag` column. Each device transmits 50 different measurements which include phase wise current, voltage, inductance, and so on, and these types get stored in the `msmt_type` column. In addition, the actual operation data for each device and `msmt_type` are stored in the measurement column along with the time stamp.

C.5 Performance

The basic performance testing of the distribution grid monitoring enabler solution is done on a Quad Core 2 CPU Power 7 machine with a single disk with a Linux SUSE 11 operating system. The data is captured through Message Broker and is inserted into Informix through multiple threads of a simple WebSphere Application Server based Java program using the `executebatch()` function of JDBC.

Example C-1 shows the core piece of the Java program that inserts the data.

Example C-1 The Java function

```
ArrayList<String> provisionTagList = new
ArrayList<String>(list.size());
   try{
      connection=getConnection();
      hsLogging.info(CLASS_NAME, METHOD_NAME, "***** Database
Connection Received...");
      connection.setAutoCommit(false);
      //create prepared statement
      pst = connection.prepareStatement(sql);
      pst_set_lock_mode = connection.createStatement();
      Iterator<MeasurementData> iterator = list.iterator();
      //executing the batch
      pst_set_lock_mode.execute(sql_lock_mode);
      while(iterator.hasNext())
          {
       MeasurementData data= iterator.next();
       String piTagName=data.getMeasurementTag();
       boolean provisionTagExist = true;
hsLogging.info(CLASS_NAME,MESSAGE_FLAG_PROVISION_TAG,Boolean.toStrin
g(provisionTagExist));
       if(provisionTagExist)
           {

List<StringMeasurementValue>value=data.getStringMeasurementValues();
         for (Iterator<StringMeasurementValue> iterator1 =
value.iterator(); iterator1.hasNext();)
           {
          StringMeasurementValue val =iterator1.next();
          String timeString = new

HSDateUtil().getFormatedDateOld(val.getTimeStamp().toString());
//setting value to prepare statement
            pst.setString(1,piTagName);
        pst.setTimestamp (2,Timestamp.valueOf(timeString));
        pst.setString(3,val.getValue());
        pst.setTimestamp(4, new java.sql.Timestamp(new
java.util.Date().getTime()));
            hsLogging.info(CLASS_NAME,METHOD_NAME,piTagName);
            hsLogging.info(CLASS_NAME,METHOD_NAME,timeString);
            hsLogging.info(CLASS_NAME,METHOD_NAME,val.getValue());
            //adding to batch
```

```
          pst.addBatch();
         }
        }
        else
            {
          provisionTagList.add(piTagName);
        }
           }
         int[] updateCounts = pst.executeBatch();
         hsLogging.info(CLASS_NAME,METHOD_NAME,"Updated Row After
Batch insert :"+ updateCounts.length);
         connection.commit();
    }
```

With this simple code, the DGME observed 33500 (5x more than expected) records per second insertion rate, consuming just 4.4 GB RAM and hardly 50% CPU utilization. The performance was observed consistently, even when the TimeSeries table had 3 months of data stored in it. The avenue open to further improve this performance by more than 10x, with even lower resource utilization, is to have multiple discs and use Informix TimeSeries fast loader functions.

The data retrieval operations like aggregation, consecutive record comparison, difference between first and last element and finding running average were found extremely easy and fast. The 6 concurrent sessions retrieving 3 months of tag data responded in less than 1 second while in parallel the data was uploaded with high insertion rate. The data maintenance activity, such as updating statistics for consistent performance, always took less than 1 second.

C.6 Ease of development

The measurement values for each tag are stored in a TimeSeries table with every 10 min interval and 144 records inserted per day per tag. However, the dashboard showing a high-level view of substation details needs to aggregate or sample the data at higher interval levels to draw the chart to show the trend and to determine if it is needed to drill down to raw data.

This process becomes easy with the Informix TimeSeries `aggregateby` function. For example, the following example shows the average value of phase 1 current for substation `S004781`:

```
SELECT * FROM table
((
 SELECT
```

```
tssettolist
((
  aggregateby
  ('avg($value)','ts_1hour',measurement,0,
  "2012-01-02 00:00:00.00000"::datetime year to fraction(5),
  "2012-01-02 23:50:00.00000"::datetime year to fraction(5)
  )::timeseries(measurement_row)
 ))::list(measurement_row not null)
FROM measurements
WHERE msmt_tag='S004781' AND msmt_type='IL1'
));
```

This query gives average current for phase 1. As the calendar passed to `aggregateby` function, ts_1hour, follows an hourly pattern. The aggregation can be changed to any level just by changing the calendar pattern, as shown in Figure C-2.

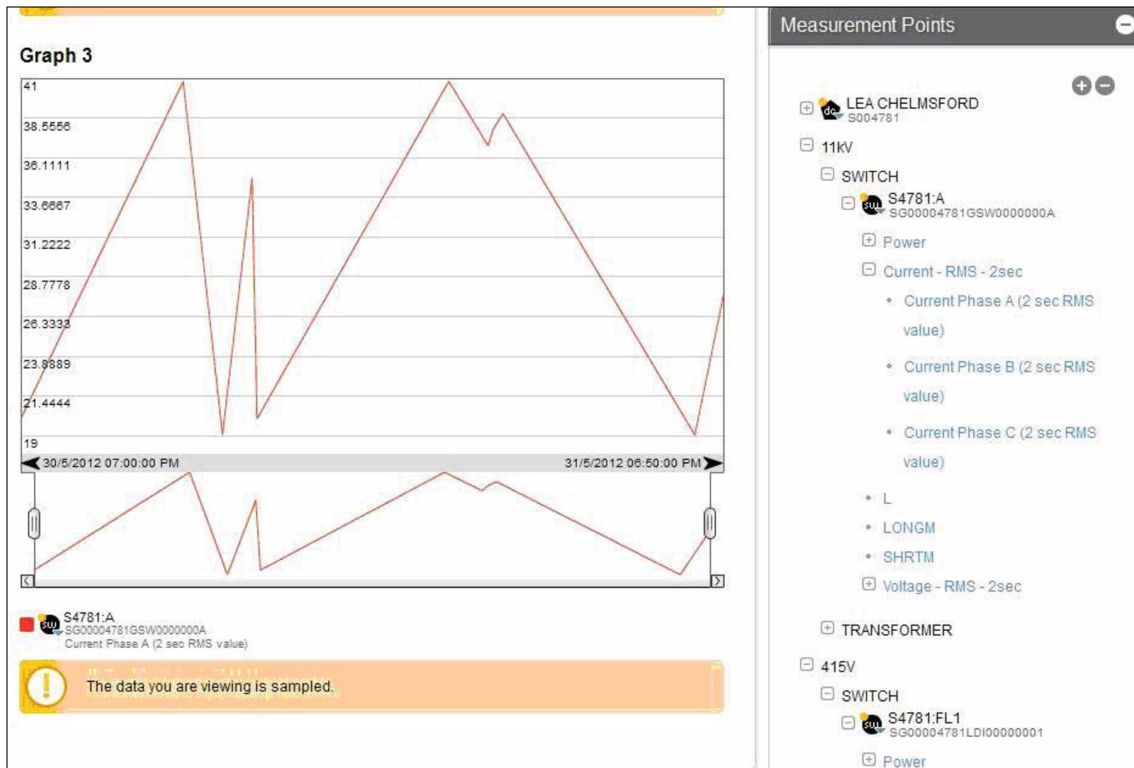


Figure C-2 Average current for phase 1

The substation utilization is another critical functionality that helps to determine overall utilization of the grid and to take action on under- and overutilized

substations. The overall utilization of a substation at any instance is decided on maximum current drawn on any of the three phases of the lower voltage side of the distribution transformer.

This value is calculated as a derived tag value during every insertion of substation operational data. The utilization percentage is calculated as a percent ratio of max(IL1,IL2,IL3) and fixed rating of corresponding transfer. This basically means to:

- Fetch the latest element of measurement type IL1,IL2 and IL3 of substation tag.
- Derive maximum value.
- Find the percent ratio with fixed rating stored in the ratings table.

These things can be achieved by the following simple query:

```
SELECT max(row1.value)/rating(msmt_tag) * 100 FROM
(
 SELECT getlastelem(measurement) row1
 FROM measurements
WHERE msmt_tag="S004781" AND msmt_type in
 ("IL1","IL2","IL3")
)group by 1;
```

In the code, rating (`msmt_tag`) is the user-defined function that brings the rating of the transformer to which the `S004781` tag belongs from PNM.

While showing the consolidated view for all substations on a Google map, a similar query for all substations can run under a chosen zone, as shown in Figure C-3.

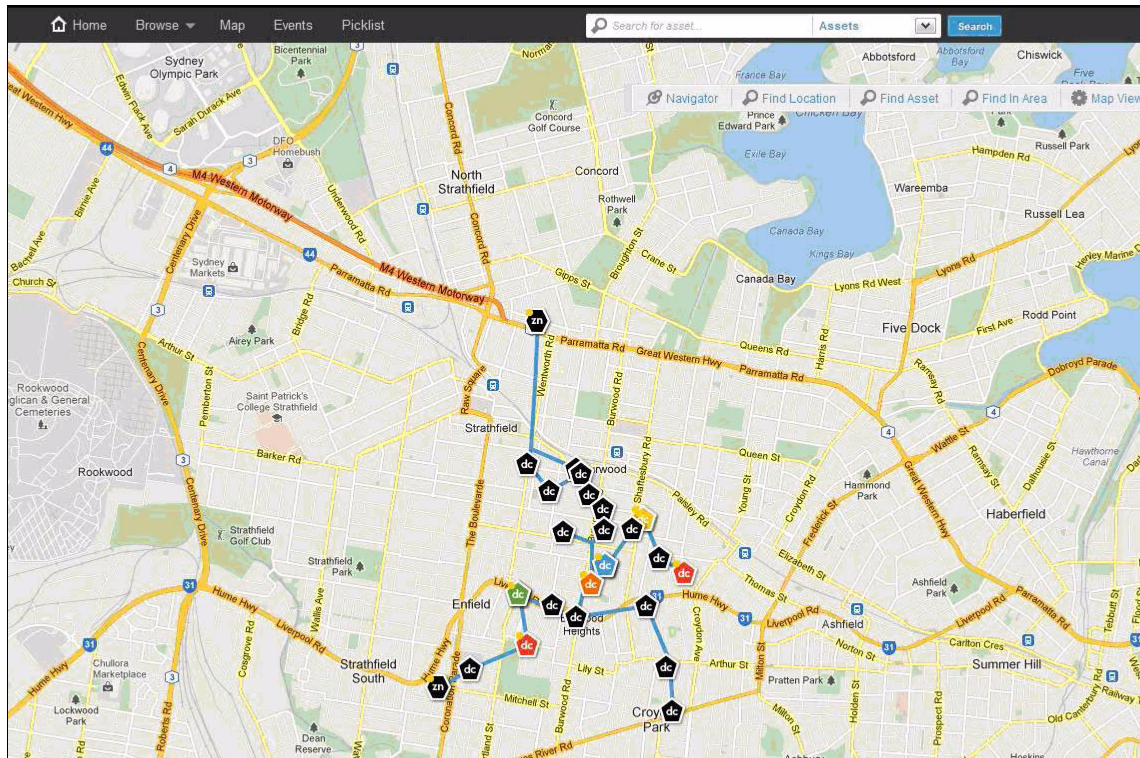


Figure C-3 Substation utilization

The color coding indicates the utilization level of each substation.

C.7 Summary

The distribution grid monitoring enabler solution is a unique solution that enables the real-time monitoring of distribution grid operations. It can help improve overall efficiency of utility companies. Although the solution has challenges, it is the best technology to address this challenge. The TimeSeries feature of Informix efficiently manages operational data by providing high performance, efficient real-time data capture, space reduction by over 50%, and ease of development. Its relational capabilities manage the power network model to provide context to the operational data. Informix built-in routines in the SQL layer can make application development easier and more efficient.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks publications

The following IBM Redbooks publications provide additional information about the Informix product. Note that some publications referenced in this list might be available in softcopy only.

- *Informix Dynamic Server 11: Advanced Functionality for Modern Business*, SG24-7465
- *Customizing the Informix Dynamic Server for Your Environment*, SG24-7522
- *IBM Informix Developer's Handbook*, SG24-7884
- *Embedding IBM Informix*, SG24-7666
- *IBM Informix Flexible Grid: Extending Data Availability*, SG24-7937

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Other publications

These publications are also relevant as further information sources:

- *Administering Informix Dynamic Server, Building the Foundation, ISBN 978-1-58347-076-3*

© Copyright IBM Corp. 2012. All rights reserved.

Online resources

These websites are also relevant as further information sources:

- IBM Informix 11.70 Information Center

  http://publib.boulder.ibm.com/infocenter/idshelp/v117/index.jsp?topic=/com.ibm.po.doc/new_features.htm

- Informix smart meter central (wiki)

  https://www.ibm.com/developerworks/mydeveloperworks/wikis/home?lang=en#/wiki/Informix%20smart%20meter%20central/page/Welcome

- Informix product family

  http://www-01.ibm.com/software/data/informix/

Help from IBM

IBM Support and downloads

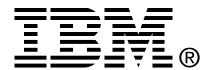
ibm.com/support

IBM Global Services

ibm.com/services

Solving Business Problems with Informix TimeSeries

(0.2”spine)
0.17”<->0.473”
90<->249 pages



Solving Business Problems with Informix TimeSeries

Greatly reduce storage requirements for time-based data

Simplify processing with built-in and custom functions

Speed up loading, access, and retrieval of data functions

The world is becoming more and more instrumented, interconnected, and intelligent in what IBM® terms a *smarter planet*, with more and more data being collected for analysis. In trade magazines, this trend is called *big data*.

As part of this trend, the following types of time-based information are collected:

- Large data centers support a corporation or provide cloud services. These data centers need to collect temperature, humidity, and other types of information over time to optimize energy usage.
- Utility meters (referred to as *smart meters*) allow utility companies to collect information over a wireless network and to collect more data than ever before.

IBM Informix® TimeSeries is optimized for the processing of time-based data and can provide the following benefits:

- Storage savings
- Query performance
- Simpler queries

This IBM Redbooks® publication is for people who want to implement a solution that revolves around time-based data. It gives you the information that you need to get started and be productive with Informix TimeSeries.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-8021-00 ISBN 0738437239