RC25327 (CHI1211-001) November 5, 2012
Computer Science

IBM Research Report

Next Generation Real Time Operational Database by Extending Informix

Sheng Huang, Xiao Yan Chen, Kai Liu, Yao Liang Chen, Chen Wang
IBM Research Division
China Research Laboratory
Building 19, Zhouguancun Software Park
8 Dongbeiwang West Road, Haidian District
Beijing, 100193
P.R.China

Simon David
IBM Informix Software Team

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Next Generation Real Time Operational Database by Extending Informix

Sheng Huang, Xiao Yan Chen, Kai Liu,
Yao Liang Chen, Chen Wang
IBM China Research Lab
{huangssh@cn.ibm.com}

Simon David
IBM Informix Software Team
cosmo@uk.ibm.com

ABSTRACT

In the era of "the Internet of Things", more and more applications face the challenge of how to manage the massive volume of data generated by various sensors and devices in the current data management systems. Realeal time databases handle the data with operational technology (OT) characteristics (high volume, long lifecycle, simple format). However, while achieving excellent "write" performance, these systems provide limited "read" capabilities. In this paper, we present a new real time operational database (RODB) system. Our system addresses the "read" problem by extending the anatomy of the Informix system architecture. The core ideas allow complicated queries in SQL manner to deal with various advanced "read" tasks while keeping the "write" advantages of the existing real time databases. We demonstrate the high efficiency of our system on both "write" and "read" applications with a variety of real case studies in the domains of C&P, E&U and facility data management.

Keywords

Relational DB, Real Time DB

1. INTRODUCTION

As sensors are becoming ubiquitous and inexpensive, the sampling data from sensors are attributed a large share of today's archive spaces in big data management. There are all kinds of data format in sensor based applications such C&P, E&U, facility management, transportation and healthcare. Among them, there are huge volumes of data generated in the format of scalar data or time series data, especially in C&P, E&U and facility management applications. There are two typical scenarios. In the first scenario, the number of sensors is not large, but the sensors are with high sampling rate (10HZ). A typical oil detection application, for example, can have a thousand of sensors collecting data simultaneously with a sample rate at about 500HZ for each. The other scenario, in contrast, may involve massive sensors with low sampling rate (<1HZ). A Smart Grid system in one province may possess 10,000,000 sensors with a 15-minute sampling interval for each sensor. Both scenarios need to store historical data for long lifecycle, process real time query and do analysis over historical data.

Although traditional relational databases (RDB) like DB2, Oracle, dominate the information technology (IT) data management market for decades for their rich query capabilities, the real time databases (RTDB), like PI Server, which have excellent performance to "write" in real time, occupy the main market of operational technology (OT) data management. However, besides the accumulated OT data, a typical OT application also relies on the other business data stored in relational database together for deep analysis, auditing, planning and optimization, etc. Thus, RDB and RTDB always play together in such applications, leading to redundant investment in data management systems and huge computing overhead for data migration from RTDB to RDB for analysis.

In this proposal, we present the next generation real time operational database system RODB. For "write" performance, we design a new storage component inside Informix, using data compression and buffer technology to reduce the I/O and storage size to meet real time persistence requirement. For "read" capabilities, we extend the Informix Virtual Table Interface (VTI) to support native SQL queries directly to the compressed data. Although the related technologies have been studied in academia, to the best of our knowledge, we are the first to combine two aspects to build a database system with the follow key features:

High write throughput. Support high write throughput with up to 1M data points/s for both high frequency sensor and massive low frequency sensors. Balanced read/write optimization on data store enables high query performance.

Compressed data store. After buffer/write/compression processing, the data is compressed with 10~100 compression ratio.

Transparent data Store. Data are viewed as simple (*id*, *timestamp*, *value*) virtual table while the internal data store structure, and data organization are transparent to users.

RTDB/RDB Fusion. Both relational data and operational data are stored in a single database. Unified data access interface (SQL) is provided to support data extraction and fusion from both operational data and relational data.

In the rest of this proposal, we will give a brief description of the system architecture to fulfill the design goal. In the final presentation or full paper, we will give a detailed introduction about the design of the system with comprehensive experimental analysis.

2. SYSTEM TAXONOMY

2.1 System Architecture overview

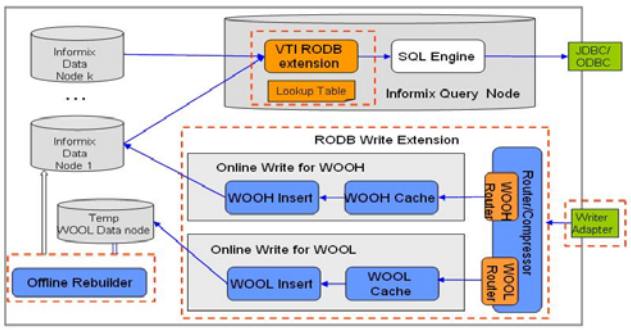

Figure 1 RODB System Architecture Overview

Figure 1 illustrates the overview of real time operational database (RODB) System Architecture where the extended parts are highlighted by dashed line. A writer adapter is used to receive the data from SCADA/Sensor environment. Then, the router/compressor will compress the data according to error acceptance and dispatch the data to different buffer pools based on the data characters. The high frequency data will be packaged and written to the Informix data node directly in a special data store format called WOOH. In contrast, the low frequency data will be written in WOOL format to a temp data node and periodically rebuilt into ROO format for persistent storage. A detailed description of the three data formats is given in section 2.2. The data will be partitioned horizontally by time to different data nodes. The partition information will be stored as meta data in a lookup table inside the Query Node. When a SQL query comes, the Query Node will lookup the meta table and then extract the required data from one or more data nodes with the help of the VTI extension.

2.2 Data Store

The main challenge for RDB to play as the historian of operation data is the low write throughput since the RDB could typically process only several 10K records per second. One reason is that the relational DB has no native support for time series data. A straightforward solution is to store each data point as a (*id*, *timestamp*, *value*) tuple, where *id* indentifies different *data source*s (i.e. different attributes of the sensors). However, this solution leads to redundant storage costs on both *id*s and *timestamp*s.

In RODB, we proposed a novel data store structure, Write Oriented Optimized (WOO) structure, to alleviate the I/O pressure. The basic idea is to reduce the number of records and, accordingly, the index size. As shown in Figure 2, there are two variances of WOO. One is WOOH structure for High frequency sensors and the other is WOOL for Low frequency sensors. When a new data point comes, the buffer pool packages the data point into a WOOH or WOOL structure. The WOO structure will not be inserted into the database until the ValueBlob field is fed enough number of *value*s (*Count* for WOOH and *GroupSize* for WOOL). The number of records is thus reduced from the total number of data points *N* to *N*/*Count* for WOOH and *N*/*GroupSize* for WOOL case, respectively. As a result, the storage costs of *timestamp* are dramatically reduced on both the data itself and the related index.

In addition, as a WOOL structure may contain data points from different data source *id*s, the query performance can be considerably low when extracting data from WOOL structures with a specified data source *id*. To address this problem, the Read Oriented Optimized (ROO) data structure is designed, where data points are re-grouped by *id* and adjacent *timestamp*s. In run time, a maintenance job will be fired periodically in background to rebuild WOOL data into ROO structure.

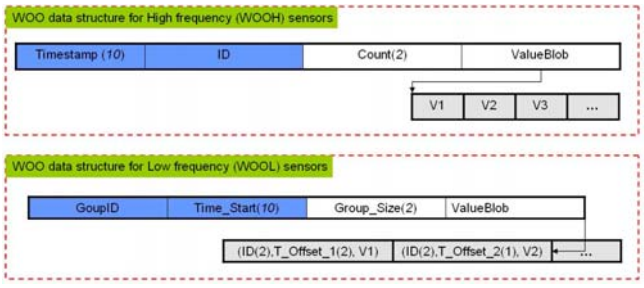

Figure 2 Data Store Structure

In the real implementation, quantization compression (high frequency vibratory data) and linear compression (smooth data) will be applied in data store for 10~100-fold data compression with some acceptable error bound. Due to the space limitation, we leave the detailed buffer/write/compression/data distribution mechanism and the corresponding data store structure variances to be introduced in the full presentation.

2.3 Data Query

This feature separates RODB from all the other real-time databases like PI server. While traditional real time databases only provide crude query functionality, RODB, with the power of SQL, enables complicated analytic job to be done with small efforts. Moreover, a pure SQL manner dramatically lowers the cost to migrate an existing system from relational database to RODB.

To integrate our query engine seamlessly, we adopt the Virtual Table Interface (VTI) extension as a data gateway. VTI is an extension to the Informix DataBlade API that allows users to develop their own *primary access methods*, consisting of a group of *purpose functions*. It can be used to access legacy data that is not stored in an Informix database as if it were an Informix table. Figure 3 shows a purpose function call flow from Informix when a SELECT query is executed.

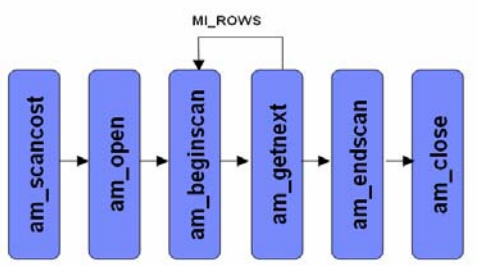

Figure 3 Processing a select statement scan

Before a query is passed to a virtual table (VT), Informix will decompose it and filter all unrelated elements including projections (described in *scan descriptor*s) and conditions (described in *qualification descriptor*s) that do not belong to the VT. A typical query processing involves the following steps: *am_scancost* first assesses the cost of the requested scan for the optimizer; *am_beginscan* interprets the scan and qualification descriptors; *am_getnext* scans the table to select and return rows that satisfy the query; *am_endscan* frees the resources allocated during the scan; *am_open* and *am_close* do the initialization and cleaning respectively. In RODB, for each data type, a VT is generated and exposed to users, which contains all the tuples whose values share the data type. Then an internal view is created, correspondingly for each VT, from the meta data to hold all data sources (indentified by *id*) that generate *value*s with the specific data type (denoted as *Type View*). For example, all the events with *Integer* values will be composed of a VT *rodb_vt_int*. Its corresponding Type View *rodb_int_view* will also be created, which contains all data sources that generates Integers. By this way, we enable users to select data from multiple data sources using one simple SQL. For example, the following SQL gets all *Integer* data from the data sources in area 'S1': *select * from rodb_vt_int a, sensor_meta b where a.id=b.id and b.area='S1'*.

Following, we give a brief introduction on the implementation of our query engine based on VTI. The three main purpose functions *am_scancost*, *am_beginscan* and *am_getnext* are described. To

avoid confusion, we use the term “record” to denote a data row in VTs to differentiate from a data row in physical RODB tables.

When a query is passed to a VT, the first task is to estimate query costs in *am_scancost* purpose function. A set of qualification descriptors are passed in for precise cost estimations. Each qualification descriptor points to one column of the VT (i.e. *id*, *timestamp* or *value*). The costs are simply estimated as the potential number of the rows that will be selected during the query. This number is basically approximated by a “SELECT count(*)” clause in combination with a WHERE clause generated from the qualification descriptors pointing to *id* and *timestamp*. The generated SQL is performed on one or more related physical RODB tables, receptively, with a join to the VT’s Type View to further filter unqualified *id*s. The related physical tables are identified by looking up the meta data table with a WHERE clause generated by the qualification descriptors pointing to *timestamp*. Different TLV data types lead to different count methods in physical tables. Qualification descriptors pointing to *value* are ignored in this phase since *value* is stored in a compressed form in the physical tables and cannot be accessed by simple aggregations.

With the cost estimations, Informix will determine an overall query plan. *am_beginscan*, *am_getnext* are responsible to take the plan as told. In *am_beginscan*, the related physical RODB data tables are located by the same way as described in *am_scancost*. Then the query is redirected to these tables to identify all rows with qualified *id*; all rows with unqualified *timestamp* interval are filtered. A data row in the physical table typically contains multiple records in the VT, sometimes only a part of the records in the same row are with qualified *timestamp*. Such row should not be filtered. Finally, *am_getnext* will extract all the physical rows one by one, in a pipeline manner, and translate each row into the output records. Filter criteria on *timestamp* and *value* are applied to the output records to make sure no unqualified records go out. Due to the space limitation, we will describe the detailed implementation of our query engine in the full presentation.

3. SUMMARY

In this paper, we present a new real time database to address the problem of volume and velocity event/motion data management in smarter planet environment by extending the Informix database server. The system packages and compresses the data in its own format, with the parallel writing technology, the write and read performance can be improved 1-2 order of magnitude than relational database to cover the typical usage scenarios in C&P, E&U, facility data management, etc. In the final presentation, we will give a detailed description of the implementation and typical usage scenarios benchmark result sharing.