Views, Objects, and Persistence for Accessing a High Volume Global Data Set

Richard T. Baldwin
US DOC NOAA
National Climatic Data Center
Rich.Baldwin@noaa.gov

Abstract

Efficient access methods are reviewed and explored in relation to the global surface hourly data set and several of its derivative products. Typical access paradigms are compared for object persistence using streamed output files, key/value databases, and object and relational databases (ODBMS/RDBMS). An overview of Java Data Objects (JDO), Enterprise Java Beans (EJB), and other alternatives to persist objects are given. Efficiencies gained by implementing RDBMS views or aggregates are also investigated. Given spatial, temporal, and analytical factors, a conservative estimate for the number of data objects from global hourly data approaches 2 billion.

1. Introduction

Part of the mission of the National Climatic Data Center (NCDC) is to archive and provide access to world and US weather data. There are many different types of data archives at NCDC totaling well over 500 terabytes. This paper will focus on a data set of particular interest to many of our customers in industry, government and the military. The Integrated Surface Hourly (ISH) data set is comprised of observations, generally recorded hourly for a group of up to 40 different weather elements. The observations are from US military, civilian, and global stations, some starting before 1900 [1]. The database totals approximately 350 gigabytes and contains records for nearly 20,000 stations worldwide as shown in Figure 1. These data are archived on a Hierarchical Data Storage System (HDSS), a tape robotics system, and are currently being loaded into an Oracle database.

A weather element in the ISH data set is any particular type of weather observation (temperature, visibility, wind speed and direction, etc.). There are 40 of these types of elements recorded in ISH (not all of theses elements are present at every observation). Additional derivative elements can be calculated from these elements (eg. relative humidity). From these elements, summarizations can also be calculated which isolate particular trends of interest. Currently NCDC and the US Navy are working to produce a set of ten summaries to be made available on-line. The number of these summaries will at least double over the next year. An example of a climate summary would be a table of wind speed versus direction over a 30-year period for a given station. This particular summary would be of use to the Federal Aviation Administration (FAA) in airport design and to the armed forces in mission planning. Other summaries would also include present weather, temperature, relative humidity, etc. ISH summaries can be thought of as objects, entities which are uniquely contained. Objects other than summaries can also be derived from ISH weather elements. These could include observations which are event specific or combinations of one or several additional data sets. An example of an observation specific object would include hourly temperature, minimum and maximum temperature, daily average temperature, heating degree days, etc. An event specific object from a hurricane, tornado or blizzard would include a whole host of pertinent weather element observations.

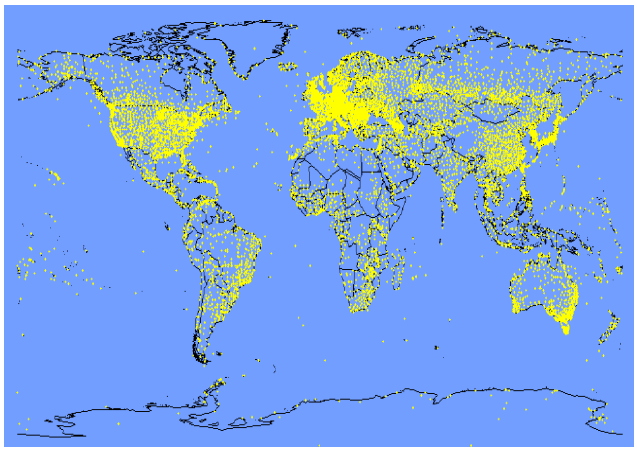

Figure 1. ISH global data distribution

2. Object Persistence

A goal for data managers is to be able to retain and quickly access the unique information defined by a specialized object whether that is described as a summary or some other combination of information. There are two broad options available for persisting objects; 1) local or remote storage in a file system or database RDBMS or ODBMS or 2) constructing an object within the context of a view or materialized view in an RDBMS. The option chosen depends on the complexity of the object and the need to store the entire object in its original form (e.g. Java Collection) or whether the object needs to be decomposed into database specific types. If an object can be adequately described in the context of a view, a mechanism to acquire that object still remains, but it should be faster.

2.1 Storage and retrieval mechanisms

The options for storing objects persistently include a simple filesystem (stream output files), a key/value database, a relational database management system (RDBMS), or an object database management system (ODBMS). Using Java, the RDBMS solution can be approached from several different specifications or application server environments, Java Database Connectivity (JDBC), Enterprise JavaBeans (EJB), Java Data Objects (JDO), or "JDO-variant" Castor. The strengths and failings of each option will be presented.

Using Java to write streamed output files to a "flat" filesystem is fast and convenient, but without any mechanism to manage efficient retrieval for a large number of files/objects. These files are portable, but this process is undoubtedly cumbersome (consider managing 1000 serialized files created from 1000 objects). Stream files can be generally viewed as limited to the implementation platform. This approach might be best limited to small implementations.

A key/value database such as DB Berkeley provides a simple, fast, and portable means of storing and retrieving Java objects without decomposition of the object. Any Java collection can simply be serialized and stored with an appropriate key. DB Berkeley is an open source solution which can be deployed on many Unix platforms. The DB Berkeley key/value database has a capacity ranging from 2 to 275 terabyte of storage depending on the page size selected for the application. The size of this database is also limited by the file size supported on a filesystem (450 gigabytes). The key/value database is single threaded, so simultaneous access is a limitation. Writing to disk occurs 10 to 15 times faster than a streamed output file [2]. The key/value database can be accessed from multiple platforms, re-hosted to other machines (e.g. into a secure SIPERNET network environment), or loaded into an RDBMS.

Object Database Management Systems (ODBMS) are storage solutions for objects used predominately in gaming. Within the database, the representation of the object is hidden from the user. Some ODBMS solutions will allow ad hoc queries of the ODBMS using ObjectQuery Langauage OQL, but these tend to be inefficient [3]. Many of these solutions are vender specific generally lacking any adoption of standards (e.g. Object Data Management Group (ODMG)). As an example of an ODBMS, consider the open source project Ozone. It is completely Java based, relatively fast, and portable. The Ozone project uses an object server that allows developers to build object-oriented, Java based, persistent applications. Java objects can be created and run in a transactional database environment. There is no object decomposition as in EJB or JDO. Ozone is compliant with W3C DOM implementations which allows for the storage of XML data. Ozone relies on a single-instance architecture where there is only one instance of a database object inside the database server, controlled with proxy objects. Since the proxy object represents the actual database object, it is used by the client as if it was the actual database object [4]; thus, the proxy effectively manages all concurrency (table integrity during simultaneous I/O) issues related to updating the object. Ozone will also allow Java objects to be stored in a local or remote object-oriented database.

Most Java implementations use JDBC as the mechanism to retrieve and store data from a RDBMS. This requires knowledge of SQL and details of the table layout and type. Implementing persistence brings problems with managing multiple tables, tracking objects and the rows and columns which they map to in a table (impedance mis-match), type differences, and portability [3]. Although this approach presents several difficulties for large problems, it is a straightforward solution for simple persistence examples. It does provide a mechanism for objects to be stored with or without decomposition through serialization and storage as a LOB (Locator Object) type. As RDBMS persistence problems become larger, additional layers like EJB and JDO are drawn upon to address these complexities. The two main issues to be mindful of are design and concurrency. A key feature of a solid persistence design or architecture is the separation of business logic and persistence logic. RDBMS persistence depends on a persistence delegate, code that hides or abstracts the details of object and table while maintaining table concurrency. A good delegate provides the benefits of optimistic concurrency with the security of pessimistic concurrency [5].

Enterprise Java Beans (EJB) is an interface layer implemented on top of a JDBC/RDBMS which provides a consistent method of presenting persistent data application

components that can be shared across many simultaneous remote and local client connections [3]. BMP bean managed persistence has largely been replaced by CMB container managed persistence as the more efficient implementation [6][7]. EJB requires a high degree of supporting software which manages database concurrency. Methods providing the object mapping or object decomposition to the RDBMS are required either through JBDC and SQL or EBQL. Managing concurrency and inefficiencies are the major issues effecting EJB which is still the preferred option of Oracle and IBM.

EJB is object relational (type) specific where Java Data Object (JDO) is persistent type neutral. The JDO specification to date has been implemented by a number of vendors which have led to dependencies. Two prominent RDBMS vendors are not supporting the JDO specification claiming that it lacks RDBMS feature portability (e.g. sequences etc.). While this is true and probably a minor point, these vendors are pushing their own solutions. Software development for JDO involves writing setter and getter methods for the elements of an object and an XML mapping file for object decomposition. The object-to-database translation code is dynamically built from these resources. This technology will allow the storage of a Java object without having to manually tear a collection down into primitive types or strings, and without the need for any SQL code to ingest the data. This approach will work with any relational database which has a JDBC driver (Oracle, MySql, etc.). JDO allows for multiple transactions and effectively merges the data model and the object model of our storage/access problem [3]. The tedious aspect of JDO is the construction of the mapping file.

The open source project Castor is a JDO variant which effectively handles concurrency issues, maps Java types to RDBMS types using XML, and maintains portability within RDBMS solutions. Castor provides a Java framework to build objects which can be mapped in to an RDBMS. By using xDoclet plug-ins with Castor , a developer needn't worry about mapping Java types to database types (using O/R mapping tools like Cocobase, Jdeveloper/TopLink, etc.). The mappings are handled automatically. Castor dynamically builds the object-to-database translation code. When requiring object decomposition, Castor is an effective open source solution, however it lacks clear documentation and does not build cleanly.

2.2 Database Views

Views or materialized views are used in data management to increase the speed of queries on a very large database. Queries to large databases like ISH involve joins between many tables, which are very expensive operations in terms of time and processing power. Materialized views improve query performance by pre-calculating expensive join and aggregation operations on the database prior to query execution time[8]. A materialized view is simply a specialized view where the database manages the indices, aggregation, etc. as additional data are added to the tables contained in the view. The concepts of managing large tables, views, and materialized views are associated with either transactional databases or redundancy data warehousing. Along with this is the notion of the "data mart" [9][10]. A "data mart" is a smaller component of the data warehouse in both content and time. A database design goal is to explore ways that data can be separated into "data marts" to improve accessibility from the transactional (current production system) database, while including the necessary infrastructure to maintain information useful in a data warehousing environment.

3. Discussion

The immediate task of deploying ISH summaries into the Climate Data Online (CDO) system (http://cdo.ncdc.noaa.gov) raises a few questions. How can a series of relatively complex summarizations be represented as views or materialized views with a RDBMS? If views are impractical, how can the summary be represented as a persistent object without requiring decomposition? Which storage method provides the best throughput: a streamed I/O filesystem, key/values database, RDBMS, or ODBMS?

Views can provide a significant cost savings (eg. 10-fold) for large tables. The current ISH operational database volume is limited to 2 years while the development database contains 5 years of data. The final operational system will have 40 plus years of data. It may be that once all of the data are loaded for ISH a cost benefit for a materialized view with aggregation is realized, but currently, the optimal approach is to include aggregate functions in queries for the appropriate summaries. This approach is further supported by the difficulty in applying an aggregate (averages, standard deviations) for summaries by hour per month for any number or combination of years potentially requested by users.

As this database evolves, the applicability of "data marts" may be realized. Conceptually, ISH materialized views can be represented in a star diagram where subsets of the database are created (Figure 2.) With ISH, materialized views of specific weather elements can be generated and then accessed with much greater efficiency than a query against the entire database table. Likewise specific weather events could also be represented by views which join multiple tables from data sources other than ISH. So, the "data marts" could represent both objects of weather elements and combinations of weather elements for a narrowly focused event.

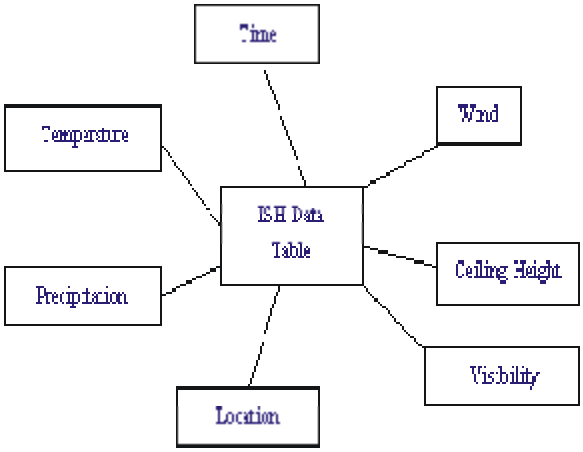

Figure 2. Star model diagram for ISH.

The EJB, JDO, and Castor solutions are all interesting paradigms; however, each decomposes an object into database types. This is not immediately needed for ISH summaries but may be of interest later in defining derivative ISH products. As a test, a prototype system was built using OpenFusion (JDO) The prototype defined a weather station object which contained various ISH weather elements (temperature, wind speed, wind direction, and precipitation). A persistence object was created which managed the station object's persistent transactions with the relational database (the setter and getter methods). The JDOEnhancer uses an XML descriptor file to map the elements of the station object in to the database. The JDOEnhancer also built the SQL code to create and populate tables in the database. Assembly and compilation of this implementation was quick and straightforward. Insertion and modification of elements in the weather station object were accomplished without any complications.

For the remaining persistence mechanisms (stream output file, key/value database, JDBC/RDBMS, and ODBMS), prototypes were built and tested. A prototype summary for wind speed and direction was created in a Java hash table. This object was then persisted through these storage mechanisms. Table 1 shows the results of object loading and access times. Each object or summary is relatively small (2-3 Kb). The key/value database showed the best overall performance. The RDBMS solution using Oracle showed the poorest results. To further test the performance of the key/value database, wind summaries for approximately 9500 ISH stations were processed by year for 1995 through 1999 resulting in the creation and storage of nearly 48,000 objects in a database file of 196Mb. Access times for objects stored in this large database were not less than those times recorded below.

Table 1. ISH transaction times.

	Load	Fetch
Stream File	21 msec	12 msec
Key/Value	2 msec	23 msec
ODBMS	45 msec	65 msec
RDBMS	180 msec	39 msec

Another performance metric is shown in Figure 3 where the size of an object is compared to the load and fetch rates for the persistent mechanisms. Generally, the transfer rates increase with larger sized object. This possibly can be attributed to a lower percent of overhead (opening or locating an file/object) effecting transfer rate as the object increases in size. It is interesting to note that the key/value database performs, comparatively, much better with smaller objects. The fetch rate surpasses the load rate as key/value objects increase in size. Also note that performance for the ODBMS database peaks with objects at approximately 100kb. Overall the key/value database showed the best consistent performance of all the mechanisms reviewed.

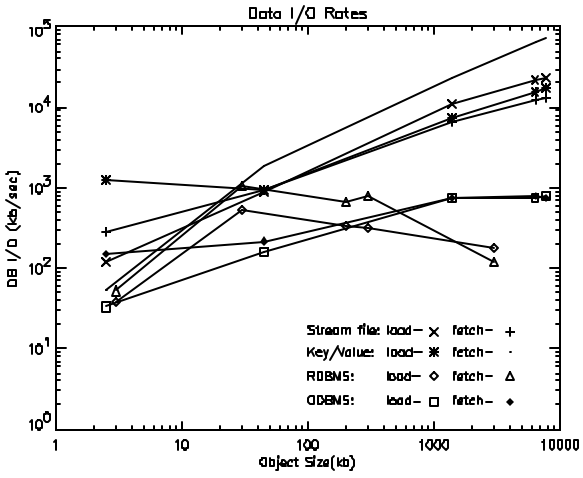

Figure 3. A comparison of data transfer rates versus the size of an object.

4. Conclusion

This overview of persistence mechanisms and views provides current methods of storing and accessing data objects. The test cases have laid the ground work for further ISH software and database development. A critical issue for data access in a global environment can be summarized

by determining what system configurations are best suited for rapidly deploying the desired data products. For ISH, the solutions; materialized views, aggregates, key/value databases and JDO, each play a role. These ideas will continue to be expanded upon as our understanding of this technical problem evolves.

Acknowledgments. Many thanks to Neal Lott for document review and proofing, Dee Dee Anders for DBA tips and pointers, and to Jeff Duska for keeping me straight on all of the Java persistence mechanisms.

References

[1] N. Lott, R. Baldwin, P. Jones, The FCC Integrated Surface Hourly Database, A New Resource of Global Climate Data, *US DOC/NOAA/NCDC Technical Report 2001-01*, 2001

[2] Berkeley DB, http://www.sleepycat.com

[3] R.M. Roos, *Java Data Objects*, Addison-Wesley, London, 2002.

[4] F. Braeutigam, G. Mueller, P. Nyfelt, *Project Ozone: Users Guide*, http://www.ozone-db.org

[5] G. Reese, *Database Programming with JDBC and Java*, O'Reilly & Associates, 2nd edition, Sebastopol, CA, 2000.

[6] J. Walker, Using Bean-Managed Persistence in EJB Entity Beans, *Oracle Magazine*, vol. XVI, issue 4, pp. 70-76, 2002.

[7] J. Walker, Implementing CMP in EJB Entity Beans, *Oracle Magazine*, vol. XVI, issue 6, pp. 92-97, 2002.

[8] Oracle 9i Materialized Views, *white paper*, (Oracle Corp.2001),http://otn.oracle.com/products/oracle9i/pdf/o9i_mv.pdf

[9] W.H. Inmon, E. Young, What is a Data Mart?, *white paper*, (Pine Cone Systems Inc., 1997), http://63.170.41.42/library/whiteprs/techtopic/tt04.pdf

[10] W.H. Inmon, Data Marts and the Data Warehouse: Information Architecture for the Millennium, *white paper*,(Informix,1999),http://www.billinmon.com/library/whiteprs/infx_dm.pdf