Multi-threaded Query Agent and Engine for a Very Large Astronomical Database

Aniruddha R. Thakar, Peter Z. Kunszt, Alexander S. Szalay and Gyula P. Szokoly

Center for Astrophysical Sciences, The Johns Hopkins University, 3701 San Martin Drive, Baltimore, MD 21218; Email: thakar@pha.jhu.edu

Abstract.
We describe the Query Agent and Query Engine for the Science Archive of the Sloan Digital Sky Survey. In our client-server model, a GUI client communicates with a Query Agent that retrieves the requested data from the repository (a commercial ODBMS). The multi-threaded Agent is able to maintain multiple concurrent user sessions as well as multiple concurrent queries within each session. We describe the parallel, distributed design of the Query Agent and present the results of performance benchmarks that we have run using typical queries on our test data. We also report on our experiences with loading large amounts of data into Objectivity.

1. Introduction

The Sloan Digital Sky Survey (SDSS) is a multi-institution project to map about half of the northern sky in five wavelength bands from ultraviolet to infrared (Szalay 1999, this volume). It is expected to image over 200 million objects and collect spectra (redshifts) for the brightest 1 million galaxies among these. The raw data is expected to exceed 40 Tb, and the resulting science archive will exceed several Tb. The processed and calibrated data will be stored in the SDSS Science Archive (SX). The data in the SX is expected to be a few Tb in size, and will be accessible to the entire astronomical community through specialized tools. A commercial ODBMS, Objectivity, is used as the data repository.

The astronomical community will interact with the SX using a graphical user interface (GUI). The SX GUI allows users to construct queries formulated in the *SX Query Language* (SXQL) using pull-down menus and graphically selecting cuts in coordinate space. SXQL is a simple SQL-like language that implements the basic subset of clauses and functions necessary to formulate queries for an object database. The SX GUI connects to the SX Query Agent over a socket and allows the user to build and submit SXQL queries to the agent.

2. The SX Query Agent

The SX Query Agent is the *sxServer* daemon, which authenticates users and maintains multiple concurrent user sessions. It also allows multiple concurrent queries per user session. The sxServer software is fully multi-threaded. It incorporates multi-threading at several levels. At the top level, there are two main threads - a socket listener that accepts new connections, and a thread that cleans up expired sessions. Each new user session in turn spawns two threads - one each for input and output. Each query is then executed in multi-threaded mode by the sxEngine library module, as described below. Searches on remote partitions in a distributed federation are executed in parallel remotely by the multi-threaded *Remote Thread Server* (sxRTS).

Each query is first analyzed in order to provide a projected "query cost" estimate. The projected query cost is computed by first building a query tree, and then intersecting the query tree with a pre-constructed multi-dimensional index tree (Kunszt et al. 1999, this volume). The query cost is specified in terms of the subset of the database that must be searched, and a rough time estimate for the search. The user can decide based on the cost whether the query is worth running or not. Once the query is launched, the query execution tree is executed by the *Query Engine*, which returns the objects that are selected by the query. The individual fields within each object desired by the user are then extracted from each object by the *Extractor*. Output from queries is routed back to the user (GUI) by default but can be directed to an *Analysis Engine* (AE) instead. A suite of AEs will be available to facilitate science with the SX.

2.1. The SX Query Engine

The sxEngine library module implements the query engine, which executes a query and returns a bag of matching objects. We adopt the concept of a bag from OQL. A bag is simply an unordered collection of object-pointers (Object-IDs or OIDs in Objectivity jargon). A distinct bag is therefore a set.

2.2. SX Query Execution Tree

The input to the Engine is the *Query Execution Tree* (QET). Each node of the QET is either a *query* or a *set operation*. The currently supported set operations are Union, Intersection and Difference. The supported query primitives are:

- **Scoped Query** - A subset of the federation is specified for the search in the form of a nodelist. The predicate is applied only to selected type of objects.
- **Bag Query** - The Bag Query applies the given predicate to each object in the bag that it obtains from its single child node.
- **Association Query** - This query applies the given predicate to all objects that are linked to each object in the given bag.
- **Proximity Query** - This is a search for all objects that are nearby in space to a given object. Such a query is very useful and common in astronomy.

The scoped queries form the leaves of the QET, and interior nodes can be any other type of query or set operation. *Each node of the QET is executed in a separate thread*, and an ASAP data push strategy (*data from a child node is pushed up the tree using a stack as soon as it becomes available, rather than waiting for the child node to finish*) ensures rapid response even for long queries.

There are a number of other modules that play an important role in the SX machinery. The Agent Pool library module maintains a pool of *agents* (agent = thread containing an Objectivity context). The Parser module is responsible for parsing the SXQL query and converting it to a query tree, which is then passed to the Intersector. The Intersector intersects the query tree with the Spatial and Flux Indexes (see Kunszt et al. 1999). The Partition Map tells the query agent how the data is distributed in the federation by identifying partitions that are on local and remote partitions. This enables data on different partitions to be searched in parallel.

2.3. The SX Abstract

The sxAbstract library module provides a run-time abstraction of SX data model. It allows manipulation of DB objects without knowledge of schema, retrieval of data values, and invocation of methods. In effect, it behaves as a metadata server or type manager. The Abstract can be used by any application/module that uses a given data model.

2.4. The SX Extractor

The sxExtractor library module provides the functionality needed to extract individual or groups of attributes (members) of a given object (OID). It therefore executes the SELECT ... part of the SXQL statement. The Extractor uses the Abstract to retrieve data values, execute member functions and follow association links to arbitrary depths.

2.5. Port Daemon

The Port Daemon ensures that the GUI and Analysis Engine communicate with the sxServer on the correct port and also performs process-level authentication.

2.6. Data Loader

This is the application that loads new data into the Objectivity federation. SDSS data-updates will occur infrequently at well-defined intervals, and hence the updates will be done offline by a separate application. During the testing and commissioning phase of the SDSS, the data that needs to be loaded is only a few to tens of Gb in size. Even so, we have experienced loading times of the order of several hours. The presence of many association links in the data degrades the loading performance considerably. In addition, we have noticed different commit times between architectures. The total commit times in real time are given in Table 1. A database with 350k objects is loaded under identical conditions on an SGI IRIX Release 6.2 and an DEC Alpha Digital-UNIX 4.0. On both systems, the hard-drives are locally mounted. Under usual circumstances, the Alpha outperforms the SGI by a factor of 2. The same data is loaded on both machines, committing after 10k and 20k objects. The first two Alpha columns show that the commit time on the Alpha is much longer than it should be. If a special loading procedure is used on the Alpha, where the program exits and is restarted after every 30k commits, then we get the expected commit time on the Alpha (last column) - about 35 minutes. One would expect that the total commit time would scale linearly with the number of objects. This is in fact true on the SGI, but on the Alpha, the commit time seems to be correlated to the total number of objects loaded so far. Since the identical C++ code was running on both machines, this cannot be due to a program error, and the times

Table 1. Total commit times on different platforms.

SGI 10k	SGI 20k	Alpha 10k	Alpha 20k	Alpha 20k (restart)
1:08:26	1:10:54	3:45:35	3:11:27	0:37:03

are measured by timing the commit() call only. The exit-and-restart procedure gives the expected behavior on the Alpha. It is as if Objectivity fails to clean up properly after a commit on the Alpha.

3. Objectivity Issues

3.1. Performance Issues

We have run several benchmarks with our test SDSS data to gauge Objectivity performance in terms of the number Mb of data searched per second. Table 2 shows the results from two different types of queries that were run on the DEC-Alpha platform with Objectivity version 5.1. The first type of test is a straight search on all "tag" objects in the federation, i.e. small descriptor objects that are $\sim$ 100bytes in size and are used for fast searching and indexing. The second test is a straight search on all photometric data objects, which are each $\sim$ 2k in size. The federation had been "tidied" to reduce fragmentation. The raw sequential I/O performance for the RAID storage media was $\sim$ 11 Mb/s. The total number of objects (of each type) in the federation was 1.1 million, with 63 databases and 64k containers. Each of the two tests was run in two modes: locking (write-locks) and non-locking (read-only) mode. These are labeled (L) and (NL). 3 different page sizes were tried: 8k, 16k and 32k. The last column

Table 2. Objectivity performance benchmarks on DEC-Alpha.

Page size→	8k	16k	32k	32k[C]
Tags (L)	4.2	3.2	3.7	4.2
Tags (NL)	4.4	3.8	4.4	4.6
Objs (L)	1.0	0.7	0.64	9.2
Objs (NL)	1.2	0.86	0.75	10.46

(32k[C]) is after we reduced the number of containers that our flux data was spread across. The peak speeds for the tag objects were 4 - 4.5 Mb/s for the 8k & 32k federations, and 3.8 Mb/s for the 16k federation, in NL mode. The speeds for the large photo objects were lower by a factor of 3 for all page-sizes, but increased dramatically (factor of $\sim$ 2) to 10 Mb/s after the container reorganization. Running in no-lock mode is faster by 5-10%.